DOCUMENT RESUME

ED 257 441 IR 011 673

AUTHOR	Kurland, D. Midian, Ed.
TITLE	Developmental Studies of Computer Programming Skills. A Symposium: Annual Meeting of the American Educational Research Association (New Orleans, Louisiana, April 23-27, 1984). Technical Report No. 29.
INSTITUTION	Bank Street Coll. of Education, New York, NY. Center for Children and Technology.
PUB DATE	Oct 84
NOTE	135p.
PUB TYPE	Viewpoints (120) -- Reports - Research/Technical (143) -- Collected Works - Conference Proceedings (021)
EDRS PRICE	MF01 Plus Postage. PC Not Available from EDRS.
DESCRIPTORS	Cognitive Development; *Cognitive Processes; *Computer Science Education; Computer Software; Curriculum Development; Elementary Education; Epistemology; *Learning Processes; *Programers; *Programing; Secondary Education; *Skill Development

ABSTRACT

The five papers in this symposium contribute to a dialog on the aims and methods of computer education, and indicate directions future research must take if necessary information is to be available to make informed decisions about the use of computers in schools. The first two papers address the question of what is required for a student to become a reasonably proficient programmer. The first--"Mapping the Cognitive Demands of Learning to Program" (D. Midian Kurland, Katherine Clement, Ronald Mawby, and Roy D. Pea)--reports a study of high school programming novices who participated in an intensive summer programming course. The second paper--"The Development of Programming Expertise in Adults and Children" (D. Midian Kurland, Ronald Mawby, and Nancy Cahir)--examines how expert programmers acquired their skill, with attention to the amount of time invested and the type of resources available when they were learning to program. The last three papers look beyond programming to the issue of transfer. The third--"Issues and Problems in Studying Transfer Effects of Programming" (Kate Ehrlich, Valerie Abbott, William Salter, and Elliot Soloway)--examines whether learning to program helps students solve problems in other related intellectual domains. The fourth--"What Will It Take to Learn Thinking Skills Through Computer Programming?" (Roy D. Pea)--discusses research on the transfer of high level thinking skills from programming. The final paper--"Making Programming Instruction Cognitively Demanding: An Intervention Study" (John Dalby, Francoise Tourniaire, and Marcia C. Linn)--describes a study in which a curriculum was designed explicitly to make programming more cognitively challenging. A concluding commentary by Jan Hawkins discusses the issues raised in the papers and offers thoughts on current and future directions for research in this field. (THC)

ED257441

CENTER FOR CHILDREN AND TECHNOLOGY

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☑ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official NIE position or policy.

"PERMISSION TO REPRODUCE THIS MATERIAL IN MICROFICHE ONLY HAS BEEN GRANTED BY

Denis Newman

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

1984 AERA Annual Meeting
April 23-27, New Orleans, LA

Symposium:
Developmental Studies of Computer Programming Skills

Edited by
D. Midian Kurland

Technical Report No. 29

IR011623

Bank Street College of Education 610 West 112th Street NY, NY 10025

1984 AERA Annual Meeting
April 23-27, New Orleans, LA

Symposium:
Developmental Studies of Computer Programming Skills

Edited by
D. Midian Kurland

Technical Report No. 29

October 1984

IR011673

DEVELOPMENTAL STUDIES OF COMPUTER PROGRAMMING SKILLS

Technical Report No. 29

D. Midian Kurland	1	INTRODUCTION
D. Midian Kurland, Katherine Clement, Ronald Mawby, and Roy D. Pea	2	MAPPING THE COGNITIVE DEMANDS OF LEARNING TO PROGRAM
D. Midian Kurland, Ronald Mawby, and Nancy Cahir	3	THE DEVELOPMENT OF PROGRAMMING EXPERTISE IN ADULTS AND CHILDREN
Kate Ehrlich, Valerie Abbott, William Salter, and Elliot Soloway	4	ISSUES AND PROBLEMS IN STUDYING TRANSFER EFFECTS OF PROGRAMMING
Roy D. Pea	5	WHAT WILL IT TAKE TO LEARN THINKING SKILLS THROUGH COMPUTER PROGRAMMING?
John Dalby, Francoise Tourniaire and Marcia C. Linn	6	MAKING PROGRAMMING INSTRUCTION COGNITIVELY DEMANDING: AN INTERVENTION STUDY
Discussant:		
Jan Hawkins	7	

INTRODUCTION

D. Midian Kurland

Center for Children and Technology
Bank Street College of Education

This collection of papers was originally presented at a symposium entitled Developmental Studies of Computer Programming at the 1984 American Educational Research Association annual meeting in New Orleans. The symposium was a forum for presenting research on the development of computer programming skills. While the papers were intended for a research audience, there was considerable interest in this topic by teachers and educational policy makers as well. Clearly, programming is a topic of great concern to many in the educational community. While these particular papers do not specifically address policy issues, many of the issues they do raise have direct bearing on educational practices. Thus, they should be of interest to all those concerned with the role of computer programming in the precollege curriculum.

The systematic study of how children interact with computers is only just beginning. The fields of software psychology and developmental cognitive science are still very much in their infancy. Yet schools are under pressure right now to make decisions about what they should be doing with programming. Whether because of a belief in the educational value of programming or fear of being left behind, in the past two years many schools have opted to introduce some form of programming instruction into their curriculum. In many schools it has now replaced traditional CAI as the primary way in which computers are being used at most grade levels. The determination to teach programming has occurred despite the almost total absence of information on how to teach programming, or what aspects of programming languages children of various grade levels are most able to understand and use effectively.

The rush to acquaint students of all ages and abilities with the basics of computer operation and programming has engendered a raft of new problems which extend far beyond the selection of the best hardware and software packages. While promoters of computer programming have been

quick to make promises about its educational benefits, they are promises that have not as yet been fulfilled through practice, nor addressed adequately through systematic research. Instead, the field is being driven by largely philosophical debates on the potential educational value of letting students discover science, math, and linguistic concepts through the medium of programming. The claims for the benefits of programming have excited the imagination of thousands of teachers, parents and researchers. Unfortunately, however, in moving from educational philosophy to pedagogical practice, we find ourselves on less fully developed ground. Educators are giving their students time to freely explore with the computer using languages such as Logo, or are teaching students to code in BASIC or Pascal on the grounds that programming is a fundamental skill necessary to participate fully in the new "information society". Yet because no one knows quite what to expect from such instruction, nor even how or what to assess in order to know whether the instruction is successful, programming's foothold in the schools, particularly at the earlier grades, is tenuous, and the directions it should be taking unclear.

At the center of the current uneasiness over the role of programming in schools is concern with its relationship to the the rest of the curriculum. Teachers are realizing that despite all the rhetoric about how "easy" it is for kids to program, to learn to program <u>well</u> is a difficult and time consuming enterprise. Yet without achieving at least certain minimal levels of competency, many students (as well as teachers) become frustrated and feel stymied in their efforts to use the computer in ways they feel are appropriate. However, simply allotting more time to programming instruction is not an appropriate response in most cases. Nothing that is new and time consuming can be added to the school curriculum without it replacing something else. What then should computer programming replace? For how long? For which students? At what ages? And once students have mastered the rudiments of programming, how can it be incorporated into the rest of curriculum so that it can be used in the service of other educational goals?

Pressure is mounting on policy makers to find answers to these questions. Yet at the present time there is little systematic research which policy makers can draw on to help guide their decisions. Aside from a limited number of anecdotal reports, little is known of how novices, particularly children, learn programming languages and the pragmatics of debugging and testing programs. Little too is know, despite all the claims, about the impact programming may have on the development of a wide range of high level thinking skills. While computers are rapidly becoming intricately interwoven into the very fabric of our society, just what students should know of their inner workings, and how this knowledge should be taught must for the most part be left to speculation.

The five papers in this symposium examine computer programming from a variety of perspectives. In doing so they raise a number of important issues which in turn have serious policy implications. The purpose of these papers, it should be pointed out, was not to provide answers for all the issues that they raise. Rather, they contribute to a much needed dialog on the aims and methods of computer education, and indicate directions future research must take if we are to have the information required to make informed decisions about the use of computers in schools. By providing systematic data on how children learn about programming a computer, contentions about what children can or can not do with computers and programming can be more fully constrained by actual observations of novices in the process of learning. At the same time, such data moves us closer to achieving our long range goal of creating a developmental theory of programming. Such a theory must ultimately serve as the cornerstone of any effective programming pedagogy.

The papers in this collection fall into two categories. The first two papers address the question of what is required for a student to become a reasonably proficient programmer. The last three papers look beyond programming per se to the difficult issue of transfer. Specifically, they ask whether programming promotes the development of generalizable thinking skills or mathematical concepts.

The first paper by Kurland, Clement, Mawby and Pea reports on a study of high school programming novices who took part in an intensive summer programming course. This paper examines some of the potential cognitive prerequsites for being able to program well. The second paper by Kurland, Mawby and Cahir reports on an interview study with expert child and adult programmers. This paper examines how expert programmers acquired their skill, with particular attention to the amount of time they invested and the type of resources they had available when they were learning to program.

The third paper by Ehrlich, Abbott, Salter and Soloway directly addresses the important issues of transfer. They examine whether learning to program helps students solve problems in other closely related intellectual domains. The issue of transfer is picked up again in the fourth paper by Pea who provides a discussion of research on the transfer of high level thinking skills from programming. The fifth paper by Dalbey, Tourniaire, and Linn describes a study in which they tried explicitly to design a curriculum to make programming more cognitively challenging.

Finally, Jan Hawkins provides a commentary in which she discusses the issues raised in the symposium and offers some thoughts on current and future directions for research in this field.

MAPPING THE COGNITIVE DEMANDS OF LEARNING TO PROGRAM

D. Midian Kurland, Catherine Clement, Ronald Mawby and Roy D. Pea

Center for Children and Technology
Bank Street College of Education

Introduction

Vociferous arguments have been offered for incorporating computer programming into the standard pre-college curriculum (Papert, 1980; Luehrmann, 1981; Snyder, 1984). Many parents and educators believe that computer programming is an important skill for all children in our technological society. In addition to pragmatic considerations, there is the expectation among many educators and psychologists that learning to program can help children develop general high level thinking skills useful in other disciplines such as mathematics and science. However, there is little evidence that current approaches to teaching programming bring students to the level of programming competence needed to develop general problem solving skills, or to develop a model of computer functioning that would enable them to write useful programs. Evidence of what children actually do in the early stages of learning to program (Pea & Kurland, 1984; Rampy, 1984) suggest that in current practices, programming many not evoke the kinds of systematic, analytic, and reflective thought that is characteristic of expert adult programmers (cf. Kurland & Cahir, 1984).

As the teaching of programming is initiated at increasingly early grade levels, questions concerning the cognitive demands for learning to program are beginning to surface. Of particular interest to both teachers and developmental psychologists is whether there are specific cognitive demands for learning to program that might inform our teaching and tell us what aspects of programming will be difficult for students at different stages in the learning process.

The work reported here was supported by the National Institute of Education (Contract No. 400-83-0016). The opinions expressed do not necessarily reflect the position or policy of the National Institute of Education and no official endorsement should be inferred. We would like to thank Karen Sheingold and Linda Caporael for their comments on earlier drafts of this paper.

In the first part of this paper we explore factors that may determine the cognitive demands of programming. In the second part we report on a study of these cognitive demands conducted with high school students learning Logo. The study was premised on the belief that in order for programming to help promote the development of certain high level thinking skills, students must attain a relatively sophisticated understanding of programming. Therefore we developed two types of measures: measures to assess programming proficiency, and measures to assess certain key cognitive abilities which we hypothesized to be instrumental in allowing students to become proficient programmers. The relationship between these two sets of measures was then assessed.

Issues in Determining the Cognitive Demands of Programming

One of the main issues in conducting research on the cognitive demands of programming is that the term "programming" is used loosely to refer to many different activities involving the computer. These activities range from what a young child seated in front of a computer may do easily using the immediate command mode in a language such as Logo to what college students struggle over, even after several years of programming instruction. Contrary to the popular conceptions that young children take to programming "naturally" while adults do not, what the child and the adult novice are actually doing and what is expected of them is radically different. Clearly the cognitive demands for the activities of the young child and the college student will also differ. Thus what is meant by programming must be clarified before a discussion of demands can be undertaken.

Defining programming and assessing its cognitive demands is problematic because programming is a complex configuration of activities. These activities vary according to what is being programmed, the style of programming, and how rich and supportive the surrounding programming environment is (Pea & Kurland, 1983; Kurland, Mawby & Cahir, 1984).

One consequence of the fact that programming refers to a configuration of activities is that different combinations of activities may be involved in any specific programming project. These activities include, at a general level, problem definition, design development and organization, code

writing and debugging (See Pea and Kurland, 1983). Different combinations of activities will entail different cognitive demands. For example, a large memory span may facilitate the mental simulations required in designing and comprehending programs. Or analogical reasoning skill may be important for recognizing the similarity of different programming tasks and for transferring programming methods or procedures from one context to another. An adequate assessment of the cognitive demands of programming will depend on analyses of the programming activity and examination of the demands of different component processes.

Specifying Levels of Programming Expertise

In assessing the cognitive demands of programming, specifying the intended level of expertise is essential. Different levels of expertise will entail different cognitive demands. In many Logo programming classrooms, we have observed children engaging in what we term brute force "paragraph" programming, or what Rampy (1984) has termed product-oriented programming. This style is analogous to so-called spaghetti programming in BASIC. When programming, students decide on desired screen effects, then write linear programs, lining up commands that will cause the screen to show what they want in the order they want it to happen. Students do not engage in problem decomposition, or use the powerful features of the language to structure a solution to the programming problem. For example, if a similar shape is required several times in a program, students will write new code each time the effect is required, rather than writing one general procedure and calling it repeatedly. Programs thus consist of long strings of Logo primitives that are nearly impossible, even for the students who have written them, to read, modify or debug. Though students may eventually achieve their goal, or at least end up with a graphics display they are content with, the only "demands" we can imagine for such a linear approach to programming are stamina and determination.

Thus, as a first step in determining what cognitive demands there are for learning or doing programming we need to distinguish between linear and modular programming (or between learning to program elegantly and

efficiently, in contrast to a style emphasizing the generation of effects without any consideration of how they were generated.)

The beginners' linear style of constructing programs, whether in Logo or BASIC, contrasts with modular programming, a planful process of structured problem solving. Here, component elements of a task are isolated, procedures for their execution developed, and the parts assembled into a program and debugged. This type of programming requires a relatively high level understanding of the language. Modular programming in Logo, where programs consist of organized reusable subprocedures, requires that students understand the flow of control of the language, the powerful control structures such as recursion, and the passing of values of variables between procedures. The cognitive demands for doing this kind of programming are different from the demands for linear programming, as are the potential cognitive benefits which may result from the two programming styles.

Distinguishing Between Product and Process

In assessing the demands for different levels of expertise, however, it is important that level of expertise not be equated with the effects the students' programs produce. We must distinguish product form process (Werner, 1937). We have seen very elaborate graphics displays created with entirely brute force programming. One characteristic of highly interactive programming languages such as Logo and BASIC is that students can often get the effects they want simply by repeated trial and error, without any overall plan, without fully understanding how effects are created, without the use of sophisticated programming techniques, and without recognizing that a more planful program could be used as a building block in future programs.

Furthermore, in school classrooms we have often seen students borrow code from each other and then integrate the code into their programs without bothering to understand why the borrowed code does what it does. Students therefore can often satisfy a programming assignment by piecing together major chunks imported from other sources. Though such "code stealing" is an important and efficient technique widely employed by expert programmers, an overreliance upon other people's code that is

beyond the understanding of the borrower is unlikely to lead to deeper understandings of programming. Therefore, if we simply correlate students' products with their performance on particular demands or programming proficiency measures, we are likely to find the correlations greatly attenuated.

Compensatory Strategies

This point suggests another important factor which complicates the identification of cognitive demands of programming. Any programming problem can be solved in many ways. Different programmers can utilize a different mix of component processes to write a successful program. This allows for high levels on some abilities to compensate for low levels on others. For example, a programmer may be deficient in planning skills needed for good initial program design but may have high levels of skills needed to easily debug programs once drafted. Thus it will not be possible to identify a unique set of skills which are necessary for programming. Instead, different programmers may possess alternative sets of skills, each of which are sufficient for programming competence.

The Programming Environment

The features of the programming environment may also increase or decrease the need for particular cognitive abilities important for programming. We cannot separate the pure demands for using a programming language from the demands and supports provided by the instrumental, instructional, and social environments. For example, an interactive language with good trace routines can decrease the need for pre-planning by reducing the difficulty of debugging. Similarly, implementations of particular languages that display both the student's program and the screen effects of the code side by side in separate "windows", such as Interlisp-D, can reduce the difficulty in understanding and following flow of control.

In learning to program, the instructional environment can reduce certain cognitive demands if it offers relevant structure, or can increase demands if it is so unstructured that learning depends heavily on what the students themselves bring to the class. For example, understanding the operation of branching statements of the IF-THEN-ELSE type requires an appreciation of conditional logic and an appreciation for the operation of

truth tables. If students have not yet developed such an appreciation, doing programs which require even simple conditional structure can be very confusing. However, with appropriate instruction, an understanding of how to use conditional commands in some limited contexts (such as conditional stop rules to terminate the execution of a loop) can be easily picked up by students. Thus in the absence of instruction, conditional reasoning skill can be a major factor in determining who will learn to program. However, with instructional intervention, students can pick up enough functional knowledge about conditional commands to take them quite far.

Instruction is important in other ways as well. For example, students in our experience are very poor at choosing appropriate programming projects on their own that are within their current ability, yet which will stretch their understanding and force them to think about new types of problems. They are poor at constructing for themselves what Vygotsky would describe as the "zone of proximal development" (Rogoff & Wertsch, 1984). Thus, too little guidance on the part of the teacher can lead to inefficient or highly frustrating programming projects. Yet too much teacher imposed structure can make the projects seem arbitrary and uninteresting, and thus less likely to evoke the full attention and involvement of the student. Finding the right balance between guidance and discovery will have a major impact on the kinds of cognitive abilities which the students will have to, or choose to, bring to the programming task.

Finally, the social context can mediate the demands placed on an individual for learning to program since programming, particularly in elementary school classrooms, is often a collaborative process (Hawkins, 1983). The varying skills of the collaborators can compensate for one another in producing a program. Thus groups of students may be able to produce programs that any one of them alone could not have produced. While work in teams is typical of expert programmers, it currently raises thorny assessment problems in an educational system that stresses individual accountability.

In summary, several factors complicate the identification of general cognitive abilities which will broadly effect a child's ability to learn to program. In asking about demands we must consider level of expertise, the impact of supportive and/or compensatory programming environments, and the role of instructional and social factors which interact with children's initial abilities for mastering programming.

Analysis of the Cognitive Demands of Modular Programming

One of the central motivations for teaching programming to pre-college students is to provide them with a tool for understanding mathematical concepts and for developing general problem solving skills. We believe that achieving this goal requires that students program well (Mawby, 1984). Knowing only basic turtle graphics commands in Logo can be a useful goal in some instructional contexts, for example, if a teacher wants students to explore angles or estimate distances. However, to use a language like Logo to develop an understanding of math concepts like variables and functions, and to learn problem solving techniques such as planning and modular problem solving, the aspects of Logo which involve these concepts must be exploited; Logo cannot be used merely as a substitute for a ruler, protractor, or form blocks. This means that students must become reasonably good modular programmers. They must learn to program with variables and procedures, to generate code that can be reusable, and to understand the control structure of the language. Given the importance of learning to program at this proficiency level, what are likely to be some of the cognitive abilities such programming requires? A rational analysis of the cognitive requirements of designing and comprehending modular programs suggests that, among other skills, means-ends procedural reasoning and decentering may be particularly important.

We expect that procedural reasoning ability is an important skill underlying the ability to program since programmers must make explicit the antecedents necessary for different ends, and must follow all possible consequences of different antecedent conditions. Designing and following the flow of control of a program requires understanding different kinds of relations between antecedent and consequent events, and organizing and

interrelating the local means-end relations (modules) leading to the final end. Procedural reasoning thus includes understanding conditional relationships, temporal sequencing, hypothetical deduction, and planning.

Decentration also may be important skill in programming since programmers must distinguish what they know and intend from what the computer has been instructed to execute. This is important in program construction and in debugging. The program designer must realize the level of explicitness required to adequately instruct the computer, and in debugging must differentiate what the program "should" do from what it did do. We have found that such decentering is a major hurdle in program understanding at the secondary school level (Kurland & Pea, 1984)

On the basis of this rational analysis we designed a study to investigate the relationship of measures of procedural reasoning and decentering to the acquisition of programming skill.

Method

To investigate empirically the relationship between these cognitive abilities and programming competence, we studied novice programmers learning Logo. Logo was chosen in part because of the high interest it has generated within the educational community, and in part because the Logo language has specific features which support certain important thinking skills. For example, the strategy of problem decomposition is supported by Logo's modular features. Logo procedures may be created for each subpart of a task. The procedures may be written, debugged, and saved as independent, reusable modules and then used in combination for the solution of the larger problem. Efficient, planful problem decomposition in Logo results in flexibly reuseable modular procedures with variable inputs. While the same can be true of languages such as BASIC, the formal properties of Logo appeared to be more likely to encourage structured programming on the part of the student.

Participants and Instructional Setting

Participants in the present study were 79 8th-11th grade female high school students enrolled in an intensive six-week summer program designed to improve their math skills and introduce them to programming. The goal of the program was to improve students' mathematical understanding while building up their sense of control and lessening their anxiety about mathematics. (See Confrey, 1984 and Confrey, Rommney & Mundy, 1984 for details about the affective aspects of learning to program.) Those admitted to the program were students who were generally doing very well in school and had high career aspirations, but who were doing relatively poorly in mathematics and, in some cases, experiencing a great deal of math related anxiety.

Each day, the students attended two 90 minute mathematics classes as well as lectures and demonstrations on how mathematics is involved in many aspects of art and science. Each student also spent 90 minutes each day in a Logo programming course. The teachers hoped that the programming experience would enable students to explore mathematical principles, and lead them to new insights into mathematics. The guiding philosophy of the program, which influenced both the mathematics and Logo instruction, was constructivist. This Piagetian inspired philosophy of instruction holds that a person's knowledge and representation of the world is the result of his/her own cognitive activity. Learning will not occur if students simply memorize constructions presented by their teachers in the form of facts and algorithms. Thus, students were expected to construct understandings for themselves through their direct interactions and explorations of the mathematics or programming curriculums.

The Logo instruction was structured around small classes with the students working primarily in pairs, i.e. two students to a computer. There was a six to one student-teacher ratio, and ample access to printers and resource materials. In order to provide structure for the students' explorations of Logo, the staff of the program created a detailed curriculum designed to provide systematic learning experiences involving the basic Logo turtle graphics commands and control structures. While the curriculum itself was detailed and carefully sequenced, the style of

classroom instruction was influenced by the discovery learning model of Papert (1980). Thus students were allowed to work at their own pace and were not directly accountable for mastery of specific concepts or commands. The instructors saw their primary role as helping the students develop a positive attitude towards mathematics and programming. In this respect, the program seemed by our observations to have been very successful.

The emphasis of the course was on learning to program and doing turtle geometry. While the teachers repeatedly drew attention to underlying mathematical principles at work in assignments given, they also tried to bring students to an adequate level of programming proficiency. Thus the curriculum was designed around a series of "challenges" (i.e worksheets) that students were to work through in a systematic order. These challenges included creating graphics using basic Logo primitives, unscrambling programs, predicting program outcomes, coordinating class projects to produce large-scale programs, and exploring properties of degrees, radians, and circles. It was assumed that the students would find the challenges and the opportunity to work at the computer enjoyable, and therefore largely self-motivating.

Measures

We were interested in how their level of programming proficiency would relate to the specific cognitive abilities which our analysis of the demands of programming had indicated to be potentially important for mastery. We therefore developed measures cognitive demands and programming proficiency to use in this study.

Demands Tasks

Two demands tasks were developed and administered to students at the beginning of the program. The first, procedural flow of control task, was designed to assess students' ability to use procedural reasoning to follow flow of control determined by conditional relations. Students had to negotiate a maze in the form of an inverted branching tree (see Figure 1). At the most distant ends of the branches were a set of labeled goals. To get to any specific goal from the top of the maze students had to pass through "gates" at each point where a branching occurred. The

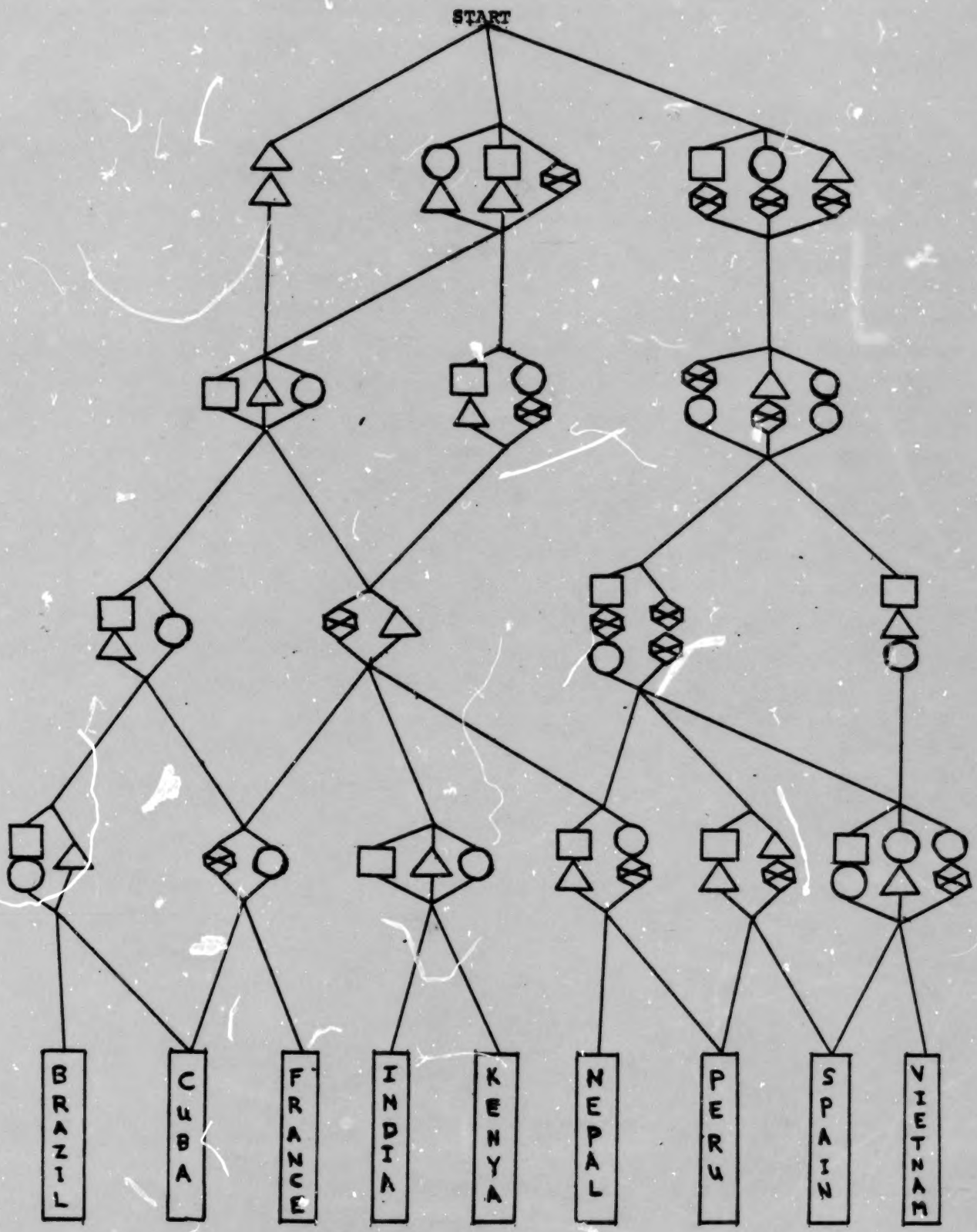
conditions for passage through the "gates" required satisfying either simple or complex logical structures (disjunctive or conjunctive). Passage through gates was controlled by varying sets of geometric "tokens" that the student was presented with at the beginning of each problem. Each gate was marked with the type or types of tokens that were required to gain passage through it. For example, if students were given a circle token, they could pass through a circular gate, but not a square gate. If they had both a square and triangle token, then they could pass through a joint square-triangle gate but not a joint square-circle gate.

The task had two parts. In the first, students were presented with five problems in which they had to find paths through the maze that did not violate the conditions for passage through the gates. Students were given sets of tokens and asked to discover all possible goals that could be reached with that set.

In the second part of the task, we designed two problems, based on a more complex maze, to add further constraints and possibilities for finding the optimal legal path to the goals. Unlike part 1, at a certain point in the maze students could choose to trade tokens of one kind for tokens of another. In addition, as they passed through each gate, they lost the token that they used to get through it. This feature introduced additional planning and hypothetical reasoning requirements as the students had to foresee the sequential implications for choosing one path over other possible paths. This task allowed for several possible solutions that met the minimum requirements of the task (i.e. reaching a specified goal at the end of one branch). However, some possible solutions were more elegant than others in that they required the use of fewer tokens. Thus it was of interest to see whether students would choose to go beyond an adequate solution to find an elegant one.

The task was designed using non-english symbolisms so that verbal ability and comprehension of the "if-then" connectives would not be a confounding factor. In natural language, if-then is often ambiguous, and its interpretation dependent on context. We did not include standard tests of the if-then connective in propositional logic because computing

FIGURE 1


START
BRAZIL
CUBA
FRANCE
INDIA
KENYA
NEPAL
PERU
SPAIN
VIETNAM

truth values as these tests require is not strictly relevant to following complex conditional structures in programming.

The procedural flow of control task, therefore, involved a system of reasonable, though arbitrary and artificial rules to make it analogous to a programming language and to prohibit subjects' use of prior world knowledge. The nested conditional structure of the tree and the logical structures of the nodes were designed to be analogous to logical structures in computer languages.

The second demands task was designed to assess decentering as well as procedural and temporal reasoning. In this debugging task students were required to detect bugs in a set of driving instructions written for another person to follow. Students were given a set of written directions, a map, and local driving rules. They were asked to read over the directions and then by referring to the map catch and correct bugs in the directions so that the driver could successfully reach the destination. Students had to consider means-ends relationships and employ temporal reasoning to follow the given instructions, determine their accuracy, and correct "buggy" instructions. They had to decenter, making a distinction between the their and the driver's knowledge, in order to tell whether instructions were sufficiently explicit and accurate. The kind of bugs students were asked to find and correct included:

(a) Inaccurate information bug: instructions given were simply incorrect (e.g., telling the driver to make a right hand turn at a corner instead of a left).

(b) Ambiguous information bug: instructions were not sufficiently explicit to enable the driver to correctly make a choice between alternative routes (e.g., telling the driver to exit off a road without specifying which of two possible exits to use).

(c) Temporal order bug: one line of instruction was stated at the wrong time (e.g., telling the driver to pay a token to cross a toll bridge before saying where to purchase the tokens).

(d) Bugs due to unusual input conditions, and embedded bugs in which obvious corrections fail because they introduce and/or leave a bug

(e.g., telling the driver to make a detour in response to a rush hour traffic rule, but failing to note that the obvious detour violates a second traffic rule).

Programming Proficiency Tasks

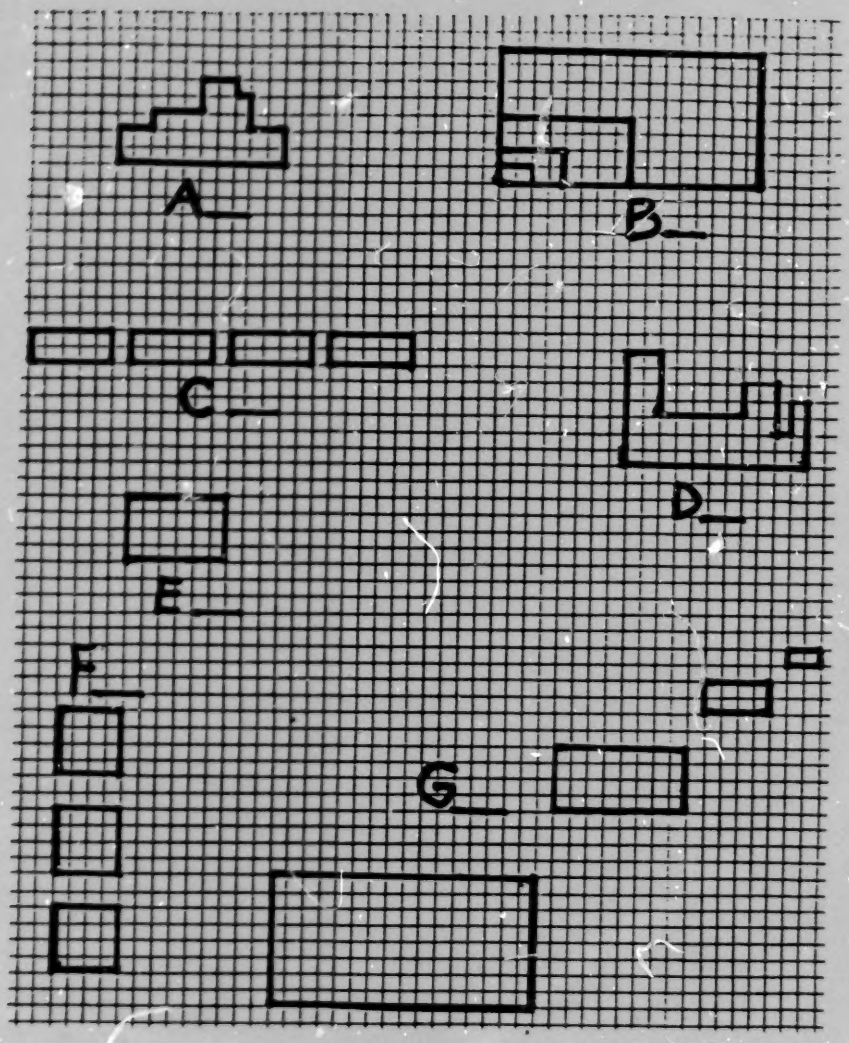
We were interested in determining the demands of modular computer programming. Therefore we needed to develop measures that would provide a detailed assessment of programming proficiency which entailed understanding of flow of control, program decomposition, and reusability of code. In designing the test we were less concerned with assessing students' knowledge of individual commands than with comprehension of the overall structure of the language and the pragmatics of programming. The test consisted of three parts: one production task and two comprehension tasks.

Production task. The production task was a paper and pencil test designed to assess planning, problem decomposition, and features of programming style such as the conciseness and generality of procedures. Students were shown a set of seven geometric figures, represented in Figure 2.

The students were instructed to select five of the seven figures and write Logo programs to produce them. The task called for students first to indicate the five figures they would write programs for, and then to number them in the order the programs would be written. This instruction, it was hoped, would encourage the students to plan before they began to write their programs. Students were free, however, to alter their choice of figures and/or order once they began to code. For each of their five programs, they were to write the code and give the run command needed to make the program produce the figure.

The task sheet included an area labeled "workspace", analogous to the Logo workspace, in which students could write procedures to be called by their programs. The layout of the task sheet, two sample problems, and explicit instructions made it clear that procedures, once written in the "workspace," were available to all programs.

FIGURE 2



The task was designed to encourage planning for modular procedures which could be reused across programs. In fact, figures B, C, E, F, and G could be programmed by writing just three general purpose procedures. An optimal solution would be to write a procedure with two variable inputs to produce rectangles, a "move over" procedure with one input, a "move up" procedure with one input, and then to use those three procedures in programs to produce figures B, C, E, F, and G. Also, Figures B and G could be most efficiently produced using recursive programs, though recursion was not necessary.

Figures A and D were included as distractor items. Unlike the other five figures, they were designed not to be easily decomposed and cannot be easily produced with code generated for any of the other figures.

The task, then, could be solved by planful use of flexible modules of code. It could also be solved in many other ways, such as writing low level, inelegant "linear" code consisting of long sequences of FORWARD, LEFT and RIGHT commands, thereby never reusing modules of code. We were particularly interested in this style dimension since a "linear" solution gives no evidence that the student is using the Logo constructs which support and embody high level thinking.

Comprehension Tasks. Each of the two comprehension tasks presented four procedures: one superprocedure and three subprocedures. The students were asked first to write functional descriptions of each of the procedures, thus showing their ability to grasp the meaning of commands within the context of a procedure. Then the students were asked to draw on graph paper the screen effects of the superprocedure when executed with a specific input. To draw the screen effects students had to hand simulate the program's execution, thus providing a strong test of their ability to follow the precise sequence of instructions dictated by the program's flow of control.

In the first comprehension tasks the superprocedure was named TWOFLAGS and the subprocedures were CENTER, FLAG, and BOX. Figure 3 presents the Logo code for the procedures and a correct drawing of the screen effect of TWOFLAGS 10.

The second comprehension task included procedures with two inputs and a recursive procedure with a conditional stop rule. The task was designed to make the master procedure progressively harder to follow. The superprocedure was named ROBOT and the three subprocedures BOT, MID, and TOP. Figure 4 presents the Logo code, and correct drawing of the screen effects of ROBOT 30 25.

Both programming comprehension tasks were designed as paper and pencil tests that did not require the use of the computer. Students were given a sheet that listed the programs, a sheet on which to write their descriptions of what each procedure would do, and graph paper on which to draw their prediction of what the program would do when executed.

Procedure

The demands measures were administered to the students on the first day of the program along with a number of mathematics, problem solving and attitude measures. (See Confrey, 1984 for a discussion of the attitude measures.) Students were tested together in a large auditorium. Instructions for each test were read by the experimenters who monitored the testing and answered all questions. Students were given 17 minutes for the procedural reasoning task and 12 minutes for the debugging task.

In the final week of the program, the students were administered the Logo proficiency test. Testing was done in groups of approximately 30 students each. Again the experimenters gave all instructions and were present throughout the testing to answer students' questions. Students were given 30 minutes for the production task and 15 minutes each for the comprehension tasks.

Results

Programming Proficiency Tasks

To use Logo as a tool for high level thinking one must use relatively sophisticated Logo constructs, such as procedures with variable inputs and super procedures which call subprocedures. To write and understand Logo programs using these language constructs one needs to understand something about the pragmatics of writing programs as well as developing a good grasp of Logo's control structure, that is, how Logo

FIGURE 3
First Logo Comprehension Task with Correct Drawing of the Resulting Screen Effects

LOGO PROCEDURES:

```
TO CENTER
 PENUP
 HOME
 PENDOWN
END

TO FLAG :X
 FORWARD 15
 BOX :X
 CENTER
END

TO BOX :SIDE
 REPEAT 4 [FORWARD :SIDE RT 90]
END

TO TWOFLAGS :X
 CENTER
 FLAG 15
 PENUP
 RT 90 FORWARD 20 LT 90
 PENDOWN
 FLAG :X
END
```

DRAWING OF SCREEN EFFECTS OF TWOFLAGS 10

FIGURE 4.

First Logo Comprehension Task with Correct Drawing of the Resulting Screen Effects

LOGO PROCEDURES:

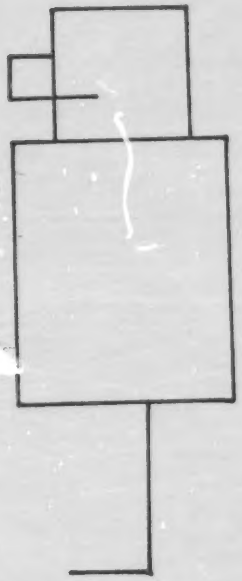
```
TO BOT :X :Y
 FORWARD :X
 RT 90
 FORWARD :Y
END

TO MID :X :Y
 BOT :X :Y
 RT 90
 BOT :X :Y
END

TO TOP :X
 IF :X = 5 RT 90 BACK 10 STOP
 REPEAT 4 [FORWARD :X RT 90]
 FORWARD 5 LT 90
 TOP :X - 10
END

TO ROBOT :X :Y
 HT
 MID :X :Y
 BACK 15 LT 90
 BOT :X - 10 :Y - 15
 RT 90 PU FORWARD 50 PD
 TOP :Y - 10
END
```

DRAWING OF SCREEN EFFECTS OF ROBOT 30 25



determines the order in which commands are executed. The empirical question addressed here is whether students develop such an adequate understanding as the result of five weeks (approximately 45 hours) of intensive Logo instruction.

Comprehension tasks. The assessments of Logo proficiency given at the end of the course indicated that mastery of Logo was limited. On the TWOFLAGS task, 48% of the students correctly drew the first flag, which required simulating the execution of TWOFLAGS through its call to FLAG in line 2. Then 21% correctly drew the second flag with 19% of the students correct on both flags, showing that in almost all cases performance was cumulative.

A third of the students were partially right on the second flag. Analysis of errors on the second flag by these students indicated that more of them had trouble following the flow of control than keeping track of the values of the variables. An error in place on the second flag suggests that the student's simulation did not execute all the positioning lines of code, especially the call to CENTER in the last line of FLAG. This reveals an error in flow of control. An error in size on the second flag suggests that the student did not correctly pass the variable from TWOFLAGS to FLAG to BOX.

On the ROBOT task, 65% of the students correctly drew the body of the robot, which involved simulating the execution of ROBOT through its call to MID. Then 37% correctly drew the leg, which involved following the execution through ROBOT's call to BOT in line 4. TOP is the recursive procedure which, with inputs to ROBOT of 30 25, would execute three times. The first time TOP draws the head, the second time it draws the nose, and the last time it draws the mouth, and then stops. Finally, 16% of the students correctly drew the head, 13% succeeded with the nose, and only 2% were able to follow the program execution all the way through to the mouth. The cumulative percentages are within 3% of these absolute percentages.

Again analysis of the errors of students who were partially correct showed that more of these students correctly passed values of variables

than followed the flow of control. In partially correct drawings, the parts of the robot were more often correctly sized than correctly placed.

The students' written descriptions of the procedures in both the TWOFLAGS and ROBOT tasks showed that many had at least a vague, general understanding of the procedures. Often students seemed to understand the code in that they gave adequate glosses of individual lines. But when tested by the drawing task, many revealed that they did not understand Logo's control structure well enough to trace the program's execution. This becomes especially clear when the order of the lines in a listing of the program differed from the order in which the lines execute.

Some students failed to grasp the fact that since variable values are local to the procedure call, values can be passed among procedures under different names. Even more failed to understand the most basic fact of flow of control: after a called procedure is executed, control returns to the next line of the calling procedure. Inadequate models of recursion, including conflation of recursion with Logo's REPEAT command, were shown by some students (cf. Kurland & Pea, 1984).

Production task. Results with the production task showed limited use of efficient programming techniques that required variables and reusable subprocedures. While most students were able to generate the figures, many did so following the "linear" programming style. Only 21% of the students avoided both distractor items. An additional 35% avoided either A or D singly. Thus 44% of the students wrote programs for both A and D. Given a low level of programming proficiency choosing the distractors is reasonable because, by design, linear programs for the distractors are easier than linear programs for figures B and G (and comparable to C and F).

Among the possible approaches to the task are "analytic" and "synthetic" decomposition. By "analytic" decomposition, we mean analyzing a single figure into component parts, writing procedures for the parts, and having the program call the procedures. By "synthetic" decomposition we mean decomposition of the entire problem set into components, writing

procedures for the parts, and then having each of the five programs call the appropriate modules of code. Note that while the five non-distractor figures contain only rectangles, the rectangles are of different sizes. Thus high level "synthetic" decomposition, unlike "analytic" decomposition, requires a general procedure with variable inputs for producing the rectangles.

Students were much more likely to use "analytic" than "synthetic" decomposition. 88% of the students at least once wrote, used and reused a procedure, thus giving evidence of some "analytic" decomposition. However, only 20% of the students gave evidence of "synthetic" decomposition by using a procedure for more than one program.

Figure 5 provides more detail on the features of Logo students used to produce the individual figures. In the analysis represented by Figure 5, we wished to know, for each figure, whether students could write code to produce it, and whether they could correctly use REPEAT, variables, and recursion. The REPEAT command is the simplest within-procedure embodiment of Logo's modular features. Variables go further in transforming procedures into reusable functions, making the procedures more general and hence more useful. Recursion is an extremely powerful Logo construct in which a procedure can call on copies of itself from within other copies. These features of Logo make modular code possible and thus support problem decomposition strategies. So in addition to the product measure of "does it run", we looked at these other indicators of programming style.

A good summary indicator of style is the number of commands used to produce the program. For these tasks elegant programs use few commands. We counted each use of a Logo primitive as one command. Each procedure call was counted as one command and on the first call to a procedure the commands within the procedure were counted. On subsequent calls to that procedure only the call itself was counted.

The graph at the top of Figure 4 shows, for each figure, the range of number of commands used, the mean, and the region containing the middle 50% of the scores. For comparison, we include the number of

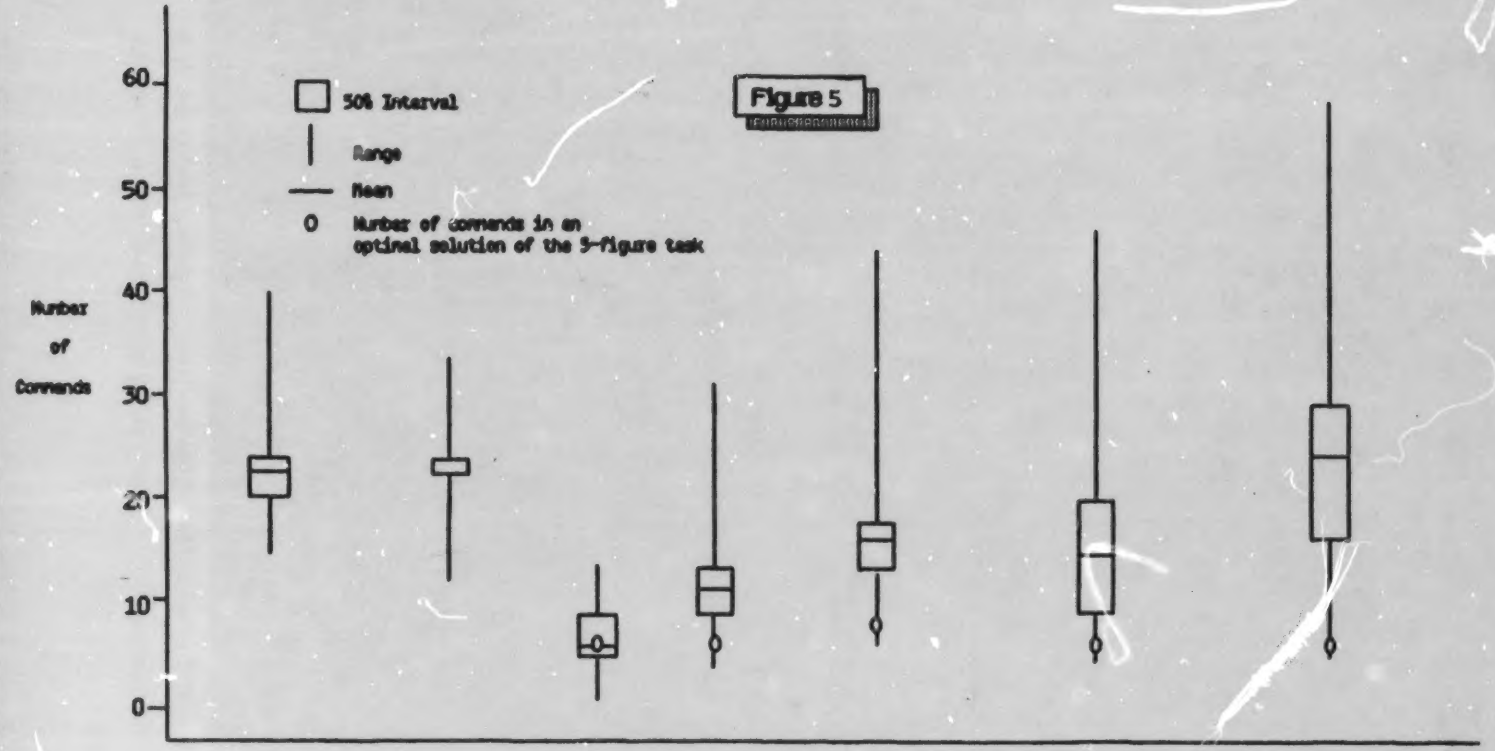


Figure 5

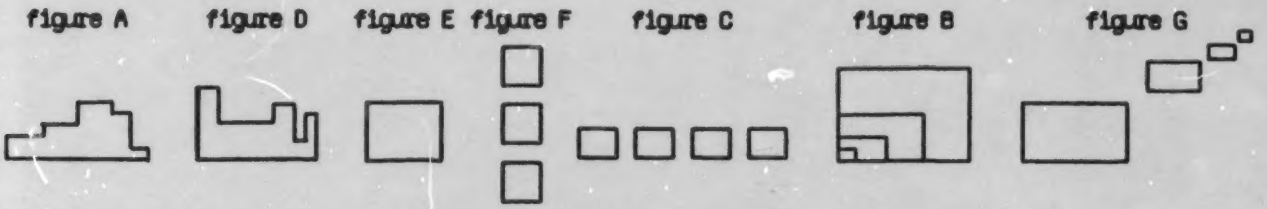


	figure A	figure D	figure E	figure F	figure C	figure B	figure G
% who did it:	.73	.51	.91	.91	.96	.60	.31
workable program:	.86	.90	.91	.80	.85	.47	.48
variables used:	.05	.02	.14	.12	.10	.43	.40
repeat used:	.08	.02	.49	.84	.65	.49	.68
recursion used:	.00	.00	.00	.00	.00	.04	.08

commands used in an optimal solution of the task as a whole. This particular optimal solution "synthetically" decomposes the five rectangular figures with three subprocedures, and produces the programs in the order E, F, C, B, G.

The figures fall into three groups: the distractors A and D; C, E, and F; and B and G. As noted, nearly half of the students chose figures A and D, and 90% of the students who chose these figures were able to write a Logo program to produce them. As expected from the design of the figures, less than 10% of these programs used variables or REPEAT. Most of the code was low level "brute force" style which could not be reused in other programs. Thus, while the students wrote programs to produce the figure, their programming style gave no indication that they were engaged in the high level thinking which Logo can support.

The group of figures C, E, and F was chosen by over 90% of the students, and nearly 90% of these students wrote workable programs for them. More than half of the students correctly used REPEAT, Logo's simpler, within-procedure modular construct. Less than 15% of these programs correctly used variables. This more elegant across-program construct was largely ignored. As a result, most students needed more than the optimal number of commands to write programs for Figures F and C.

Fewer students chose figures B and G (60% and 31%, respectively), and only half of these students wrote workable programs to produce them. These programs used REPEAT and variables more often (REPEAT: 49% in B, 68% in G; variables: 43% in B, 40% in G). It seemed that the more skilled students chose these figures and did them quite well. Of the others who chose these figures, about half the students did not attempt to use variables, and half used variables incorrectly. Again, because few students did "synthetic" decomposition, most programs had more than an optimal number of commands.

No one tried to write a recursive program for any of the figures except B and G, where recursion is appropriate. But fewer than 10% of the

students who chose figure B or G wrote correct recursive programs. This powerful Logo construct was conspicuous in its absence.

What factors may have kept these students from using the powerful and elegant features of Logo? It is unlikely that students did not notice the geometrical similarity among, for instance, Figures C, E, and F. But to do a "synthetic" decomposition of the task one needs to write procedures with variables. Moreover, coordinating subprocedures in a superprocedure requires a good understanding of Logo flow of control. Performance on the comprehension tasks showed that students had a fair understanding of individual lines of Logo code, but had difficulty in fact in following program flow of control. Thus to produce elegant Logo programs one needs to comprehend Logo variables and control structure.

Cognitive Demands Tasks

There was a fairly broad range of performance on the demands tasks (Clement, 1984). Many students showed moderate or high levels of reasoning skills as assessed by these tasks, and a few found the tasks fairly difficult.

Procedural flow of control task. The two parts of this task were examined individually. The first part included a series of problems for students to solve. Each problem posed a different set of constraints and/or goals for going through the maze. Some difficult problems required more exhaustive testing of conditions than others (i.e., the given tokens satisfied many nodes early on). Some had benefits for using alternate strategies, such as searching from the bottom up rather than top down. Performance was relatively low on the more difficult problems (30-40% correct as opposed to 55-70% correct on the less complex problems). This indicated that when many possibilities had to be considered, and there were no easy shortcuts to reduce the number of possibilities, students had difficulty testing all conditions. Doing so required careful attention to the structure of the maze and the sequential relationships between the gates.

In the second part, there were three levels of efficiency among correct routes corresponding to the number of tokens required to successfully reach the goal. Only 14% of the students on the first problem and 21% of

the students on the second problem found the most efficient route, while 41% of the students on the first problem and 79% of the students on the second were unable to reach the goal at all. Thus few subjects tested the hypotheses needed to discover the most efficient route.

Debugging task. Table 1 shows the percent of students detecting and correcting each of the four types of bugs in the task. As shown, inaccurate information and temporal bugs were easiest to detect and correct (72% to 91% success). It was more difficult to successfully correct the ambiguous instructions. Only 48% of the students were able to write instructions that were explicit enough for a driver to choose correctly among alternate routes. For the lines with embedded bugs only 21% of the subjects fully corrected the instructions; 40% caught and corrected one bug but not the other.

Results indicate that subjects had little difficulty detecting first order bugs and correcting these when the corrections were simple, for example, changing a number or direction to turn. However, when subjects had to be very explicit and exhaustively check for ambiguity and for additional bugs they were less successful.

Relationship of the Demands Measures to Programming Proficiency

Analysis of the relationship between these demands tasks and the assessments of programming proficiency yielded an interesting set of results. As can be seen in Table 2, the demands measures correlated moderately with composite scores on both tests of programming proficiency.

Examination of correlations with subscores on the programming production task showed that students' ability to write an adequate, runnable program was less highly correlated with demands measures than their appropriate use of variables, use of subprocedures within programs, or use of a minimum number of commands to write programs (one indication of program elegance).

Other subcomponents of the production task that we assumed would correlate highly with the demands measures, in particular whether students reused procedures across several programs or used recursion,

TABLE 1

DEBUGGING TASK
(% of students; N=79)

BUG TYPE	NO CHANGE	CATCH NO FIX	CATCH SOME FIX	CATCH & FIX
WRONG INSTRUCTION	.03	.06	na*	.91
AMBIGUOUS INSTRUCTION	.11	.41	na*	.48
TEMPORAL ORDER BUG	.16	.11	na*	.73
EMBEDDED BUGS	.29	.10	.40	.21

* This category is not applicable

TABLE 2

Correlations of "Demands" Measures with Measures of Programming Proficiency (N=79)

		A	B	C	D	E	F	G
Procedural Reasoning Part 1	A	--						
Procedural Reasoning Part 2	B	.34**	--					
Debugging Task	C	.38***	.27**	--				
Math Level	D	.51***	.38***	.42***	--			
Production Proficiency	E	.45***	.19*	.39***	.38***	--		
Comprehension Proficiency	F	.54***	.50***	.45***	.59***	.26*	--	
Teacher Rating	G	.30**	.20*	.22*	.37***	.26**	.54***	--

* p<.05
** p<.01
*** p<.001

were not highly correlated. However, few students engaged in either of these forms of programming. Thus, because of the floor effect it is difficult to know how to interpret this lack of a significant correlation. Interestingly, though few used the more advanced programming techniques, many students manifested high levels of reasoning skills on the demands measures. Their demonstrated logical abilities, however, were not sufficient to enable employment of sophisticated programming techniques. Other knowledge specific to the programming domain would appear to be required in addition to the underlying cognitive capacity to reason in the ways we assessed.

In general the correlations of the demands measure were higher with programming comprehension than with programming production. The design of the production task may have contributed to these findings. Students could write linear programs and still succeed on the task, and most did so. This was true even for those who at times in their class projects had utilized more advanced programming techniques from time to time. In contrast, the comprehension task required students to display their understanding of sophisticated programming constructs. Thus the comprehension task was better able to test the limits of programming novices' understanding of the language. However, a production task such as we employed may prove the better indicator of programming proficiency for students once they attain a more advanced level of ability.

We examined the relation between math achievement level (assigned on the basis of grade point average, courses taken in school, and scores on math tests administered on the first day of the program) and Logo proficiency. Math level was as good a predictor of programming proficiency as the specific demands measures taken individually. However, when math level was partialled out of the correlations in Table 2, they all remained significant at the .01 level or better, with the exception of the correlation between part two of the procedural reasoning task and program production proficiency. Thus our demands measures appear to tap abilities that are independent of those directly tied to mathematics achievement.

When both mathematics achievement and performance on our demands measures were entered into a multiple regression analysis, with Logo proficiency as the dependent variable, the multiple correlations were .71 and .52 for programming comprehension and production respectively. Thus a quarter to a half of the variability in tested programming proficiency was accounted for by mathematical understanding and specific cognitive abilities bearing a rational relationship to programming.

Discussion

The present study was aimed at identifying the cognitive demands for reaching a relatively sophisticated level of programming proficiency. We examined students learning Logo in an instructional environment that stressed self discovery within a sequence of structured activities, but with no testing or grading. Given this setting and amount of instruction, we found students for the most part managed to master only the basic turtle graphics commands and simpler aspects of the program control structure. While they gained some understanding of such programming concepts as procedures and variables, most did not develop enough understanding of Logo to go beyond the skill level of "effects generation". Thus, for example, though they used variables within procedures, they seldom passed variables between procedures, used recursion, or reused procedures across programs. Those aspects of programming requiring a more sophisticated understanding of flow of control and the structure of the language were apparently not mastered. Without this understanding students cannot use the powerful Logo constructs which engage and presumably encourage the development of high level thinking skills.

Nonetheless, we did find moderate relationships between the ability to reason in ways we hypothesized were critical for advanced programming, and performance on our measures of programming proficiency. The magnitude of the correlations indicated that the students who developed most in programming were also those who tended to perform better on tests of logical reasoning. However, our observations of the students during the course of their instruction and their performance on the Logo proficiency measures suggests that the actual writing of programs for

many students does not require that they use formal or systematic approaches in their work. Programming can invoke high level thinking skills, but clearly such skills are not necessary for students to get by in the early stages of writing programs to generate desired screen effects.

Conclusions

The field of computer education is in a period of transition. New languages and more powerful implementations of old ones are rapidly being developed, and more suitable programming environments engineered for both the new and established languages.

We can best assess cognitive demands of programming when we are clear about our goals for teaching programming, and how much we expect students to learn. However, to understand the cognitive demands for achieving a particular level of expertise, we must consider the characteristics of a specific language (such as its recursive control structure), the quality of its implementation, the sophistication of the surrounding programming environment (the tools, utilities and editors available), and the characteristics of the instructional environment in which it is being presented and learned.

Our results indicate that certain reasoning abilities are linked to higher levels of achievement in learning to program, but that most students often opt for a programming style which negates the need for engaging in high level thinking or planful, systematic programming. Thus, the demands issue remains clouded by inherent characteristics of interactive programming languages, which promote the use of a trial and error approach to program production, and the particular characteristics of the instructional environment in which learning occurs.

In conclusion, we have argued that asking what are the cognitive demands of programming is far from a simple or easily answered question. On the one hand, programming ability of one form or another is undoubtedly obtainable regardless of levels of particular cognitive skills. However, if by "learning to program" we mean developing a level of

proficiency which enables programming to serve as a tool for reflecting on the thinking and problem solving process, then the demands are most certainly complex and will interact with particular programming activities and instructional approaches.

Programming has the potential to serve as a fertile domain in which to foster the growth and development of a wide range of high level thinking skills. Studies are needed, however, on two fronts, if this potential is to be realized.

One the one hand, much more work is needed to discover what kinds of instructional environments and direction are best suited for achieving each of the many goals educators may have for teaching programing to children of different ages. We are only beginning to understand how to teach programming. In fact it still comes as a surprice to many parents and educators who read Mindstorms (Papert, 1980) too literally that programming has to be taught at all. But the unguided, free exploration approach, while possibly effective for some purposes, does not lead many students to a deeper understanding of the structure and operation of a programming language, and thus does not lead them to use or develop high level thinking skills such as problem decomposition, planning, or systematic elimination of errors.

New instructional approaches may dramatically facilitate the learning of programming and attainment of the proficiency levels at which more abstract and intensive thinking is required. Such approaches must be investigated. Also, in what ways such instruction will lessen or heighten the demands for learning to program to the levels specified by the new curricula will be interesting to monitor.

On the other hand, our ability to design more effective instruction will depend in part on further experimental work to tease apart the role various cognitive abilities play in influencing students' ability to master particular programming commands, constructs and styles. Our knowledge of the cognitive demands of operating with a language should help focus our instruction and to identify those aspects of programming which will be

difficult for students of different age and ability levels. While the relation found here between conditional and procedural reasoning ability and programming suggests some important skills, our conjecture is that at a more fundamental level, these tasks correlated with programming proficiency because they required the ability to reason in terms of formal, systematic, rule governed systems, and to operate within the limitations imposed by them. This, we feel, may be the major factor in determining whether students will obtain expert levels of proficiency. What remains to be determined is whether programming at proficiency levels below that of the expert require and/or help develop high level cognitive skills and abilities.

REFERENCES

Confrey, J. (1984, April). An examination of the conceptions of mathematics of young women in high school. Paper presented at the annual meeting of the American Educational Research Association, New Orleans, LA.

Confrey, J.; Rommney, P.; & Mundy, J. (1984, April). Mathematics anxiety: A person-context-adaptation Model. Paper presented at the annual meeting of the American Educational Research Association, New Orleans, LA.

Hawkins, J. (1983). Learning Logo together: The social context. (Tech. Rep. No. 13). New York: Bank Street College of Education, Center for Children and Technology.

Kurland, D.M. & Cahir, N. (1983). The development of computer programming expertise: An interview study of expert adult programmers. Unpublished manuscript, Bank Street College of Education, Center for Children and Technology.

Kurland, D.M., Mawby, R., & Cahir, N. (1984, April). The development of programming expertise. Paper presented at the annual meeting of the American Educational Research Association, New Orleans, LA.

Kurland, D.M., & Pea, R.D. (in press). Children's mental models of recursive Logo programs. Journal of Educational Computing Research.

Luehrmann, A. (1981) Computer literacy: What should it be? Mathematics Teacher, 74.

Mawby, R. (1984, April). Determining student's understanding of programming concepts. Paper presented at the annual meeting of the American Educational Research Association, New Orleans, LA.

Mawby, R., Clement, C., Pea, R.D., & Hawkins, J. (1984). Structured interviews on children's conceptions of computers. (Tech. Rep. No. 19). New York: Bank Street College of Education, Center for Children and Technology.

Pea, R.D. & Kurland D.M. (1983). On the cognitive prerequisites of learning computer programming. (Tech. Rep. No. 18). New York: Bank Street College of Education, Center for Children and Technology.

Pea, R.D. & Kurland D.M. (1984). Logo programming and the development of planning skills. (Tech. Rep. No. 16). New York: Bank Street College of Education, Center for Children and Technology.

Papert, S. (1980). Mindstorms. New York: Basic Books.

Rampy, L.M. (1984, April). The problem solving style of fifth graders using Logo. Paper presented at the annual meeting of the American Educational Research Association, New Orleans, LA.

Rogoff, B. & Wertsch, J. V. (Eds.). (1984). Children's learning in the "zone of proximal development". New Directions for Child Development, Number 23. San Francisco, Ca: Jossey-Bass.

Snyder, T. (1984, June). Tom Snyder: Interview. inCider, pp. 42-48.

Werner, H. (1937). Process and achievement. Harvard Educational Review, 7, 353-368.

THE DEVELOPMENT OF PROGRAMMING EXPERTISE IN ADULTS AND CHILDREN

D. Midian Kurland, Ronald Mawby, & Nancy Cahir

Center for Children and Technology
Bank Street College of Education

The teaching of high level, general purpose languages such as BASIC, Pascal, or Logo is the main focus of most precollege computer literacy courses. Although there has emerged, largely in the absence of reasonable alternatives, broad consensus that programming is a worthwhile activity for students, there is much less agreement on why this is so. Is programming to be taught as a skill worth knowing in its own right, or should it be used primarily as a vehicle for teaching problem solving and thinking skills? The problems associated with teaching programming are most acute at the elementary levels where educators are least equipped (or convinced of the need) to teach programming simply as a skill like reading or writing, yet at the same time are unsure of what using programming as a vehicle for teaching thinking skills actually entails.

The situation is complicated by the misconception being perpetrated through the mass media, and by educational visionaries of many stripes, that learning to program is easy for children. Consequently, schools are adding a programming requirement to their curriculum without a clear sense of what to expect other than that students of all ages will quickly master the pragmatics of programming and be able to write interesting and useful programs. In addition, particularly at the earlier grade levels, it is hoped that "doing" programming will enhance students' high level thinking skills, such as their ability to plan and solve problems by applying generalizable heuristics.

The work reported here was supported by the National Institute of Education (Contract No. 400-83-0016). The opinions expressed do not necessarily reflect the position or policy of the National Institute of Education and no official endorsement should be inferred. We would like to thank Roy Pea, Karen Sheingold and Catherine Clement for their comments on earlier drafts of this paper.

There is a growing awareness that things are not so simple. Learning to program a computer well is a complex process. In a recent paper, Pea & Kurland (in press) proposed that desired cognitive outcomes, such as improved problem solving ability or enhanced planning skills, would be linked to the level of programming proficiency obtained by students. This argument has been taken further by Mawby (1984) and Clement, Kurland, Mawby and Pea (1984). The basic hypothesis is that it is not enough to simply have children "do programming" since it is how one is programming, not that one is programming, that makes the difference in what is being learned. If programming is to be an arena in which high level thinking skills develop, then students must go beyond the typical novice level in which their programs are the simple one-to-one mapping of commands to their corresponding invariant screen effects. If we expect students to develop high level thinking skills from doing programming, then they must be writing programs in which high level thinking skills, such as problem decomposition and planning, will be required. It is important that students reach certain minimum levels of competency and understanding in their programming work before we look to programming as a domain that can foster the development of generalizable thinking skills. Thus, before addressing the question of how programming affects a student's thinking abilities, it is important to ask how programming ability itself develops.

What does it take for students to become reasonably competent programmers? Recent studies of novice programmers (c.f., Dalbey & Linn, 1984; Kurland, Clement, Mawby & Pea, 1984; Soloway, Ehrlich, Bonar, & Greenspan, 1982) have shown that for many students learning to program is a difficult undertaking, and many students fail to achieve even a modest understanding after one or two programming courses. However, studies of novices do not directly address the question of what is required to reach a reasonably high level of programming expertise. The problem with only studying novices in typical classroom programming courses is that so much of the variance in learning is carried by the instructional and social environments that surround the activity of programming (Krisler et al, this volume; Watt, 1984). We may observe students failing to develop an adequate understanding of programming

fundamentals, not because programming itself is difficult, but because the way it is presented or contexted is confusing, inappropriate, or misleading. Therefore, to address more directly the question of what is required to become a proficient programmer, we elected to take a closer look at people who were already expert programmers.

Our decision to study expert programmers was prompted by the need to place limits on our expectations of what levels of proficiency one should expect novice programmers to reach in their first programming course. Programming is such a new field, particularly where children are involved, that there are no established norms for what one can reasonably expect children of different ages and with different amounts of experience to accomplish. Thus by probing the developmental histories of expert programmers, we felt we could gain some insight into what to expect, and, equally impostantly, and what not to expect of students in programming classes at the precollege level. In addition, knowledge of what expert programmers had found effective in helping to learn to program could be useful to schools and curriculum developers. Such knowledge could guide decisions about what resources to provide for students in the programming classroom, and how much time to allocate to teaching and practice if the intention of the school program is for students to become proficient programmers.

The study was conducted in two parts. In the first we conducted a series of in depth interviews with a group of adult programmers. In the second, we interviewed and tested a group of programming "whiz kids" under the age of 15.

Expert Adult Programmers

Participants. Twenty adults participated in the first part of the study. Seven were graduate students (five men and two women, between the ages of 20 and 30), five were professional software game designers from an independent software company (four men and one woman between the ages of 20 and 30), and eight were commercial programmers who served as systems analysts for a large bank (seven men and one woman between the ages of 20 and 40). The seven students were doctoral candidates in

computer science, and had completed at least one year of their program. Three students had obtained an M.A. in computer science. The academic backgrounds of the game designers ranged from a high school diploma to a masters degree in Russian studies. The game designers averaged just over eight computer science courses; the range was from zero to twenty. The commercial programmers were less varied in their academic backgrounds; most had a college degree in engineering, computer science, or business, though --like the game programmers--one had taken no computer science courses.

All the participants began programming in their late teens or early twenties, except for one programmer who began at age 27 and two who began in high school. Seventeen of 20 reported that their first exposure to computers had been through programming itself. Of those remaining, two first used computers in conjunction with college laboratory work, and one began as a game player. Twelve of the 20 programmers learned FORTRAN as their first language, four started with BASIC, two with APL, and one each with Pascal and Assembler.

Procedure. All 20 participants were asked to complete a 65-item questionnaire. The questions requested information on their work and educational backgrounds, as well as childhood hobbies and interests, early programming history, programming related work experience. programming style, and their thoughts about a range of programming-related topics (e.g., what abilities they believed were important for a programmer to possess, how programming had influenced their social life, how programming fit into their long range career plans). The questionnaire included multiple choice, fill-in-the-blank and open-ended questions.

Upon receipt of the completed questionnaires, follow-up interviews were conducted with the seven graduate students and five game designers. These taped, one-hour structured interviews further explored the process by which the programmers actually wrote a program. Interview questions focused on such topics as the stages they went through when writing a program, the tools they used to help themselves program, where they got

their ideas, what they did when they get stuck, and what role other programmers and/or supervisors played in their work.

Results

A detailed report of the results from these questionnaires and interviews is available elsewhere (Kurland & Cahir, 1984). Here we would like to comment on three of the more striking themes which emerged from the interviews and, where appropriate, relate these observations of expert programmers to previous findings about novices.

First, there was little consensus among the programmers over what specific characteristics or abilities were important for learning to program. Most indicated that being logical, systematic, and curious about how a formal system operated were the most important traits required to excel at programming. However, the programmers also mentioned a wide variety of other traits or abilities that they believed contributed to becoming a successful programmer. This list included being creative, flexible, smart, personable, dedicated, planful, disciplined, organized and patient. Several emphasized that these characteristics were entering requirements, not outcomes to be expected from learning to program. Most indicated that they were logical and disciplined thinkers before they ever began programming and simply found programming compatible with their style of thinking. This view of the relationship of programming to thinking contrasts sharply with one of the major tenets of educational computing, namely that learning to program (like latin and geometry in past generations) is good mental exercise for developing logical thinking processes. The fact that these programmers claimed to have been logical thinkers prior to learning to program does not preclude the possibility that learning to program may for others help promote this style of thinking. However, it does suggest that educators must look more closely at the cognitive styles of novice programmers, and tailor instruction to take advantage of, or compensate for, students' preferred thinking styles.

A related finding concerned the relationship of prior technical ability in science or mathematics to programming. While the majority of

programmers reported having enjoyed mathematics in school, most felt that being good at mathematics per se was not important for becoming a good programmer. As one put it:

> ...I think the way you look at problems mathematically requires the same skills [as programming], i.e., reducing the English language to some sort of algorithmic language or your mathematical notation.

A similar finding has been noted by Fisher (cited in Johns, 1894) who, on the basis of a survey of programming manages, reported that the 10 most important attributes for success as a programming trainee were:

- °willingness to accept responsibility
- °thoroughness
- °persistence against obstacles
- °ability to be a self starter
- °ability to communicate
- °resourcefulness
- °responsibility to fulfill promises and commitments
- °enjoyment of the work
- °self-confidence and assuredness
- °high standards

She concluded that beyond certain minimum levels of intelligence and background knowledge, personal characteristics, attitude, and communication skills count more heavily than does strictly technical knowledge or aptitude in specific content areas.

A second common theme across the interviews was respondents consistently noted that learning to program well required substantial time and energy, even for those who greatly enjoyed programming. All those interviewed claimed to have put enormous amounts of time into their training. Estimates of the average time spent at a terminal while they were learning fell in the range of 20 to 35 hours per week. However, almost all of the programmers also reported that at some point when they were learning they went through a prolonged period of total immersion. Non-stop sessions of over 30 hours as well as 60-100 hour weeks at the computer were typical. For example, one software game designer described learning to program at college in these terms:

> "One semester in school I think I averaged about four hours of sleep a night for three months. My health was damaged, my brain was damaged."

A computer science student talked about learning to program in similar fashion:

> "Actually I work an incredible number of hours, as most computer scientists do and I work very weird hours. The thing about computer science is that it is a very consuming subject...I regularly spend ninety hours a week at the computer. It would be very convenient for me to to have a computer at home, but I'm not sure I'd do anything else...Unfortunately computer science can be so consuming that you have to make a very definite effort to have other interests."

A software game designer noted that the time required to learn to program seemed to be in excess of what was needed to gain mastery in other intellectual domains:

> "I was always struck at how much time I put into school compared to everybody else I knew...(those) who were taking ordinary computer science curriculum compared to English, it was at least double the amount of time. People were just putting in amazing amounts of time."

The amount of time expert computer programmers dedicated to learning contrasts sharply with the way computer time is allocated in schools. As recent surveys of classroom computer use indicate (Becker, 1983), more and more schools are including programming in their curriculum. However, studies of computer use in elementary schools suggest that, at most, children spend 50 hours a year with the computer, and that this time is broken into many separate episodes of short duration (Pea & Kurland, 1983). Even in high school programming courses where students may attend five 40 minute periods per week, the total number of hours students are likely to spend programming per year is still only as much as some programmers claim to have spent in a single week when they were learning. This extreme time differential raises serious questions about what the goals of programming instruction at the precollege level should be, given the amount of time schools are typically willing or able to devote to any single subject.

In addition to spending thousands of hours learning to program, the programmers also strongly favored uninterrupted marathon sessions when they did their work. All the programmers reported spending a great deal more time per session working on a programming problem than students are able to in schools. While all professionals devote more time to their discipline than do students working in the same field, the emphasis expert programmers placed on having long, uninterrupted blocks of time in which to work was nonetheless striking. They attributed this need to the fact that to program effectively requires keeping track of the meaning of current values of many variables and procedures, while also monitoring the way in which control passes conditionally between lines of code and program modules. Thus, the programmers felt that once they started on a problem it paid to keep working for as long as possible so that they would not lose track of where they were or what they were doing.

In a related study of the work habits of "super" programmers, Molzberger (1983) also reported being struck by the way expert programmers worked, and the importance for them of large blocks of uninterrupted time. For example, one of the programmers Molzberger interviewed stated that:

> "Interruptions can be unfortunate. It can take hours after a telephone call! When I am interrupted at an unfavorable time where many threads run together and I am not completely finished yet, I have to start all over."

The third striking characteristic of these expert programmers was the steps they went through in developing a program, particularly the amount of planning they reported engaging in prior to the actual coding of a program. They reported that they put a great deal of thought into how to design a program before they began the coding phase. They tended to first map out in detail a structure for the program to serve as a coding guide. None reported using flow charts to do this. They each had developed their own system based on personal preferences, given the particular type and size of program they were writing. For example, one programmer explained how he prepared to code in this way:

> "First I'd do a literature review, see how they [other programmers] did it. Then I'd sit down to write the algorithm in English language terms, in very broad structured programming. Then I'd break it down until I got it to a language I'm familiar with. Then I'd start coding."

Another programmer said that she tended to plan her assembly language programs and work out the algorithms she would need in the high level language PL/I, then use her PL/I program as a guide for the actual assembly language coding.

We have observed that many novices do little planning of this type (Pea & Kurland, 1984), and have little idea about what kinds of problems they and spend almost all of their programming time writing code. In contrast, these expert programmers viewed coding as much less central to the programming process. They reported that coding, the actual writing of programs, took only 20-25 percent of the total time they spent programming. The rest of the time was spent writing specifications for programs, planning and designing procedures and algorithms, systematically debugging and testing code they had generated, and in some cases documenting their code to help other programmers understand it. Just as knowledge of grammar and spelling is not the focus of good

writers, knowledge of the rules for a computer language clearly is not the focus of good programmers.

While planning and debugging skills were emphasized by the programmers we interviewed, they were nonetheless all highly proficient coders. Each knew at least three different languages, with one claiming to know 16. Many commented that knowing many languages was helpful, and that they used their knowledge of the strengths and weaknesses of particular languages in decisions about how to design their programs. Several programmers reported that they often would design a program in their favorite language, and then translate it into the language required by the particular application on which they were working. This technique was particularly popular with game designers who were forced to produce programs for small microcomputers which could not support the rich programming tools and powerful languages that the programmers prefered. Thus, these expert programmers appeared quite different from novices we had worked with in previous studies. First they had spent a tremendous amount of time learning to program. In addition, they now allocated their programming time in a manner that was very different from that of most novices. Rather than focusing on code generation, expert programmers spend much more time planning their programs, designing and testing algorithms, and testing their routines to be sure they worked properly under all conditions.

While expert programmers approach programming as a planful, systematic undertaking, (Pea & Kurland, 1984). Whether this is due to a difference in learning style, or is a function of experience is impossible to tell from these interviews. It seems likely, though, that if the goal of school-based programming courses is to encourage students to develop a more planful and systematic problem solving style, then the current heavy emphasis on coding should be rethought. At some point it becomes important for students to be able to code in some fashion in order to program effectively. However, what was clear from the interviews was that coding skill alone is not sufficient for becoming a programmer.

A clear implication that can be drawn from these interviews is that to learn to program well takes lots of focused time. Schools could elect to

devote substantially more time than they presently do to the teaching of programming, but more time could only be added at the expense of other school activities, and this seems neither likely nor particularly desirable. Rather, these interviews suggest that acquiring sufficient skill in current general purpose programming languages takes more time and effort than schools, particularly at the elementary level, can justify. If this is true, the question then becomes, how can we design an effective curriculum that focuses on the planning and problem solving aspects of program design and development, without requiring that students also spend excessive amounts of time in order to master coding.

Expert Child Programmers

The programmers in the first part of the study were adults, most of whom had learned to program in their late teens or early twenties. How much can we generalize from their experiences learning to program to the experience of much younger students? To find out, we next interviewed a group of six young programmers, each of whom had begun programming seriously between the ages of six and twelve.

Participants

To find students for this phase of the study, we contacted programming teachers, computer user groups, and local programming experts in the New York City area to identify children under the age of 14 who had clearly demonstrated expert programming ability.

Finding such children was not easy. Children who knew some programming or who spent thousands of hours playing sophisticated computer games were quickly identified, but children who were deeply involved with programming were much harder to find. We defined a programming expert as any child who had (1) written a commercially published program: or (2) had produced programs or utilities that others (e.g., their school or friends) were using; or (3) who had taught programming courses; or (4) in some equivalent way was clearly capable of producing software usable by others. We attempted to screen out children whose programming consisted solely of short programming exercises produced for their own amusement or interest.

We ultimately identified four children who appeared to meet our strict criteria for inclusion in the study, plus five others who appeared to be good programmers, though perhaps not quite meeting our criteria of expertise. In this initial sample were eight boys and one girl between the ages of 9 and 14. Each visited our Center for a day during which they were interviewed and given a battery of cognitive aptitude and programming proficiency tests. They also were asked to bring an example of their best work for us to examine.

We analyzed the examples of their work and asked them to write several programs for us in order to verify that they were knowledgeable programmers. During this phase of the study three of the original nine participants were dropped from the study because they appeared to lack sufficient expertise. For example, one child showed us a program he had written that looked impressive when run, but examination of the code showed that it had been written in an inelegant, brute force manner, with the exception of one routine which, it turned out, the child's tutor had provided. We also dropped a child who had experience doing a wide

variety of activities with computers, such as using Visicalc to keep his comic book collection organized. However, on closer scrutiny it was apparent that his father did most of the organizing and conceptual work while he served as helper. The third child we dropped was the one female programmer in the group. She used the computer primarily as an artistic medium for producing pictures. Her programming was restricted to fairly elementary routines in the Pilot language to generate graphic images. Thus, though these three children had some knowledge of programming and were interesting in other ways, they were clearly not expert programmers of the type required for the purposes of the present study, and so were not included in our final sample.

Results

The final group of clearly knowledgeable programmers consisted of six very bright boys between the ages of 9 and 14 (see Table 1 below) from middle to upper income homes. Though these six programmers were much more knowledgeable about programming than children in any of the school samples we had previously studied, they still displayed a very wide range of programming ability and understanding. However, all appeared to have a reasonably deep understanding of computer languages that went beyond simply knowledge about individual commands.

As may be seen in Table 1, all started with BASIC, primarily because that is what came with their home computer. Several also knew Assembler, Pascal and Pilot. In addition, all six had learned some Logo, either in school or at a computer camp. However, none claimed to like Logo for doing their personal projects. The main reason they gave for preferring to work in BASIC or Assembler was the greatly increased speed they felt these languages offered for the fast action, graphics intensive video games they wanted to program. The boys' personal judgement of the relative merits of one language versus another tended to be at the level of what it could do, not how elegantly it did it.

Table 1
Characteristics of Young Programming Experts

Child	Age	Age when started	Total hours programming (estimated)	Typical project	First language	Number of languages
1	14	11	1100	utilities	BASIC	8
2	13	8	3850	utilities	BASIC	3
3	14	12	1000	games	BASIC	3
4	12	10	550	games	BASIC	4
5	11	9	700	games	BASIC	2
6	9	6	1250	games	BASIC	2

Note. 'Number of languages' refers to the number of computer languages children reported knowing at least well enough for them to write a simple program.

To get a clearer idea of just how thoroughly the boys understood programming, we had them do several short tasks. One of the tasks was to describe in detail what a particular program we had prepared would do when executed. This short program was written in the most unstructured manner possible with embedded calls back and forth throughout. Novices whom we have studied usually have terrible difficulty with this kind of task since knowledge of individual commands is not sufficient for determining what the program as a whole will do. Novices often read the program from top to bottom without regard to flow of control and thus have no idea what the program will do. They also frequently misinterpret conditional statements which prevent them from accurately predicting what a program will do when run. In contrast, the boys in this group read the program following its control structure and thus were able to understand and accurately predict what the program would do. However, they varied in how quickly they could comprehend the program. The better programmers comprehended the program almost immediately, while the others took quite a lot of time. All six also volunteered suggestions about how the program could be improved (e.g., structure the code, avoid unnecessary GOTO's, insert error traps).

On a second task, the boys were asked to write a program to determine how many ways there are to make change for a dollar using nickels, dimes and quarters. This problem requires the ability to develop an algorithm (i.e., for testing all possible ways to figure change) plus the ability to use conditionals and stop rules. Five of the six boys handled the problem well. Three did it with no help, though again there were large differences in speed. Two needed one or two small hints on how to formulate the rule for testing whether a particular combination of coins satisfied the goal (i.e., 5*nickels + 10*dimes + 25*quarters = 100). The last child developed an interesting approach to the problem based on randomly selecting collections of coins to test, but his program failed to meet the stated goal of printing out all possible combinations. While his program would eventually, through random search, find all the combinations, he had no provision for testing whether a combination was a new one or one that had been previously found.

It was interesting to note that all three boys who succeeded without help tried to go beyond the simple requirements of the task and attempted to find an optimal solution. One found a tight algorithm that would get the answer by making the fewest number of comparisons possible. Another said that he would improve his BASIC program by writing it in assembler so that it would run faster. The third was distressed at the "slowness" of his algorithm but could not discover a faster one. This search for the elegant solution was something we have rarely encountered in novices, but was commented upon frequently in our interviews with the expert adult programmers.

Having demonstrated to our satisfaction that we had a group of unusually talented young programmers (though clearly not as proficient as the adults interviewed in the first part of the study), we then proceeded to question them about their work habits and how they learned to program. The interview results in many ways paralleled those with the expert adult programmers. First, it was clear that they devoted substantial amounts of time to learning to program. The boys averaged roughly 500±200 hours of programming a year. For the most experienced boy past five

years he had been programming. Again, this contrasts sharply with the 35 to 50 hours per year that most school programming courses can offer.

While the boys all did some programming in school (and one helped teach a course in BASIC), they all claimed that they learned nothing about programming in school since no one in school knew as much as they did. Their schools did not have the equipment, books, manuals or knowledgeable people required to help them with their programming problems. Thus they relied on the programmer's underground and/or expert programmers (sometimes peers) outside of school for guidance and assistance.

Not surprisingly, all the boys owned one or more computers where they did the majority of their programming. In each case, there was a significant older person--parent, high school tutor, adult friend--who had provided substantial early encouragement and, in some cases, instruction on a one-to-one basis. The three best programmers in the group (who were also the oldest) had spent the most time programming, and had the most access to knowledgeable experts. One boy, for example, spent hours every day in the computer center of the university where his mother worked. While there he programmed on the university mainframe with the help of the people at the center. Another was tutored by an adult friend with a background in electronics. Together they built electronic devices and then worked together on programming projects. Thus, in contrast to the situation in many classrooms, these boys began in a rich programming environment with ample support materials and, most importantly, knowledgeable experts to help guide them through their early learning.

The appeal of programming for these boys appeared to stem from several sources. One was the sense of power it provided. For example, one boy stated that he liked programming because: "Well, the feeling of power, definitely. Going into the software underground. So that if I write something good it will probably be copied and recopied for lots of people in America." Others commented that they liked the challenge and the feeling experienced when they succeeded in making a complex program

work. However, the dominant motivator appeared to be social. Programming played an important role in the social lives of all the boys. All six reported that many or most of their friends were involved with computer activities of one sort or another. Programming provided a common bond with their peers and with adults with whom they shared mutual interests. In addition, two of the boys had become involved with local computer bulletin board systems and communicating with other programmers through the their modems. One had programmed his own bulletin board system system for a VIC 20 computer and ran it several hours per day out of his bedroom. He claimed to have met most of his friends, including a current girl friend, over the modem. Thus, the boys not only found programming cognitive gratifying, it also gained them entry into a social network whose purpose went beyond simply helping each other with technical problems.

Conclusion

What conclusions can be drawn from these interviews with expert programmers? Three factors of relevance to schools have already been considered. First, learning to program well, like becoming expert in any other domain, takes a tremendous amount of time and dedication. Hundreds of hours per year were spent by both the adult and young programmers. For the young programmers, even this much time had not made them all fluent programmers. Several struggled over our assigned programming tasks and took a lot of time developing solutions. It seems clear that unless radically better methods for teaching programming are discovered, not everyone can or should become proficient in the general purpose programming languages available today.

However, just because becoming a highly proficient programmer requires a tremendous expenditure of time and effort does not mean that instruction in programming fundamentals should only be retained for an interested elite. On the basis of these interviews, a programming curriculum whose aim is to turn students into proficient programmers in the classroom time allotted would seem to be unrealistic. Yet we believe that there are ways to teach fundamental programming and computer science concepts in the normal school classroom that could serve as a solid foundation for

students' future interactions with computers, whether they continue to use computers for programming, word processing, or other information management purposes. The challenge facing educators today is to find a role for computer programming within the standard curriculum so that over the course of a student's school career, he or she will gain adequate exposure to the central computational concepts embodied in programming languages and increasingly called upon by the new generation of powerful application programs. Without such appreciation and understanding, it is difficult to see how student will become flexible users of these powerful new tools for managing information and learning.

Second, there are clearly significant, qualitative differences in the programming environments created for students in classrooms and those in which expert programmers learned and do their work. If students are to understand more about programming than simply what individual commands do, then we must find ways to get more expertise into the classroom. In addition, there need to be more support materials that can help students gain a larger sense of what programming is, how languages and computers work, and how particular classes of problems can be solved in alternate or more elegant ways.

Third, although all the adults and children were good at mathematics, they seemed to view their abilities in mathematics and programming as stemming from a more fundamental interest in logical, formal systems. Similarly, we were intrigued by the observation that many of the programmers in both age groups were accomplished musicians. We speculate that there is a particular cognitive style that makes understanding and controlling an elegant rule governed system such as music, mathematics, chess, physics or programming highly appealing for some people. While this is just speculation, it would be interesting to pursue this point more systematically in future research with novice and expert programmers.

We sensed from these programmers that programming did not teach them to think logically or approach problems in a more systematic manner. This is the way they approached problems prior to learning to program,

and thus was one of the reasons why they found programming so appealing. It remains an open question whether children who do not have this cognitive style to begin with would develop a more formal and systematic approach to problem solving as the result of learning to program. However, it seems that many novice programmers apply their preferred mode of problem solving to programming and thus do not necessarily or automatically develop new ways of thinking by virtue of their programming experiences.

It was clear that for these programmers, knowledge of programming entailed knowing how a programming language works, not just knowing the individual commands. Several adult programmers commented that learning a new language became a trivial undertaking for them since there are only a few distinct classes of languages. Once they understood how several worked, it was easy to map this understanding onto new languages. In contrast to the experts' grasp of the principles underlying a language, novices frequently know only what individual commands mean, and thus often cannot tell what a program does (Kurland, Clement, Mawby & Pea, 1984). The difference between novices and these experts lies in the expert's understanding of the relationship between commands and the rules of the language which determine the control structure of the program. Programming instruction from the very beginning might benefit from greater stress on flow-of-control and other structural issues, with less focus on individual commands and generating programs without any regard for style or elegance.

Finally, thorny policy issues surrounding the place of programming in the standard precollege curriculum are currently besetting our education system. While by no means definitive, this study coupled with previous work with novice programmers in classroom settings suggests that current orientations towards teaching programming at the precollege level may be seriously misguided or unrealistic. It is undeniable that there are fundamental programming concepts which students must understand in order to participate in the emerging information age (Sheingold, Hawkins & Kurland, 1984). Students need to grasp these fundamental concepts in order to effectively interact with computers, whether they choose to

program, do word processing, manipulate a data base, construct elaborate graphics, create musical compositions, or do any other of the myriad activities for which computers are appropriate. The question thus comes down to, what do expert programmers know that novices could know without having to themselves become expert programmers. The challenge that faces computer educators today is to identify these concepts and find ways to teach them, whether it is through programming or other computer-related activities.

REFERENCES

Becker, H.J. (1982) Microcomputers in the classroom: Dreams and realities. Report No. 310. Center for Social Organization of Schools, The John Hopkins University, Baltimore, MD.

Clement, C., Kurland, D.M., Mawby, R. & Pea, R.D. (1984, August). Analogical reasoning and computer programming. Paper presented at the Conference on Thinking, Cambridge, MA.

Dalbey, J., & Linn, M.C. (1984, April). Making pre-College instruction in programming cognitively demanding: issues and interventions. Paper presented at the American Educational Research Association Annual Meeting, New Orleans, LA.

Johns, R.P. (May 28, 1984) Don't judge a programmer by expertise alone. Computerworld, 18.

Kurland, D.M. & Cahir, N. (1984). The development of computer programming expertise: An interview study of expert adult programmers. Unpublished manuscript, Bank Street College of Education, Center for Children and Technology.

Kurland, D.M., Clement, C., Mawby, R. & Pea, R.D. (August, 1984). Mapping the cognitive demands of learning to program. Paper presented at the Conference on Thinking, Cambridge, MA.

Kurland, D.M., Mawby, R., & Cahir, N. (1984, April). The development of programming expertise. Paper presented at the annual meeting of the American Educational Research Association, New Orleans, LA.

Mawby, R. (1984, April). Determining student's understanding of programming concepts. Paper presented at the annual meeting of the American Educational Research Association, New Orleans, LA.

Molzberger, P. (1983)Aesthetics and programming. (in A. Janda (Ed), Chi'83 Conference Proceedings. Boston, MA.: ACM. (pp. 247-249).

Pea, R.D. & Kurland D.M. (1983). On the cognitive prerequsites of learning computer programming. (Tech. Rep. No. 18). New York: Bank Street College of Education, Center for Children and Technology.

Pea, R.D. & Kurland D.M. (in press) On the cognitive effects of learning computer programming: A critical look. New Ideas in Psychology.

Sheingold, K., Hawkins, J. & Kurland, D.M. Classroom software for the information age. (Tech. Rep. No. 23). New York: Bank Street College of Education, Center for Children and Technology.

Soloway, E., Erlich, K., Bonar, J., & Greenspan, J. (1982) What do novices know about programming? In B. Sneiderman & A. Badre (Eds.), Directions in human-computer interactions. Hillsdale, NJ.: Aplex.

Watt, D. (1984). Creating Logo cultures. Pre-Procedings of the 1984 National Logo Conference, 25-29.

ISSUES AND PROBLEMS IN STUDYING TRANSFER EFFECTS OF PROGRAMMING

Kate Ehrlich
Honeywell Information Systems
Waltham, Mass.

Valerie Abbott
Yale University
Dept. of Psychology
New Haven, Ct.

William Salter
Bolt, Beranek, Newman, Inc.
Cambridge, Mass.

Elliot Soloway
Yale University
Dept. of Computer Science
New Haven, Ct.

Address correspondence to:

Kate Ehrlich
Honeywell Information Systems
200 Smith St.
Waltham, Ma. 02154

This work was supported by the National Science Foundation, under NSF Grant IST-81-14840.

ABSTRACT

It is commonly believed that programming helps students develop new thinking skills that they can use to solve problems in a variety of problem-solving domains. We examined whether programming skills transfer by testing a group of college students before and after they had taken their first, introductory programming course. We also tested a group of students who were not enrolled in any programming course. The study focused on procedural skills. These skills are linked to understanding the steps that are needed to solve a problem as contrasted with simply memorizing formulae. Many students fail to adopt a procedural, active style of problem-solving. The study examined whether students can transfer some of the procedural skills they develop in a programming course to other non-programming problems. The results of the study offered some support for the transfer of procedural skills. However, the results were not conclusive due to unexpected problems with the control group of subjects. These results form the basis of a discussion of issues and problems related to studying transfer effects from programming.

1. INTRODUCTION

Educators are responding to the growing importance of computers and computer literacy by stressing the need to integrate computers and, especially programming courses, into the school curriculum. The motivation for the emphasis on computer education is two-fold. On the one hand it is firmly believed that students who can add programming skills to their other scholastic achievements have a greater chance than students without these skills of finding employment when they leave school. On the other hand, the emphasis on giving students training in programming reflects an implicit belief that programming indirectly improves students' problem-solving skills. This paper addresses some of the issues raised by the second of these goals: the educational benefits of programming. In particular, we will focus on one of the more far-reaching and contentious claims: that programming teaches students thinking skills which can be transferred to other problem-solving domains

Some of the belief that programming skills can transfer comes from the idea that programming emphasizes the "how" of problem-solving. That is, programming is believed to teach students general methods for solving problems ([7]). Although programming is a skill that is not easily acquired e.g., [1, 10, 9, 3], the difficulty of learning to program seems to reinforce the belief that those students who have successfully grasped the fundamental concepts of programming have learned some general problem-solving skills as well. Previous research on transfer (e.g., [7, 6]) offers some interesting conjectures and anecdotes to support the importance of teaching programming to young children to better position them to master more traditional math and science subjects. However, the empirical support for transfer is weak.

Programming constructs such as assignment statements provide powerful metaphors for the active process of transforming input values into output values. Moreover programming teaches students to solve problems by breaking the problem down into small components and it teaches students how to represent the steps involved in solving a problem. In these ways, programming, at least in procedural languages such as Pascal and Basic helps students develop their procedural skills. Furthermore, in earlier empirical studies we found that programmers were better able to write algebraic equations in the context of a computer program when compared to writing algebraic equations in a standard algebra context ([8, 4]). Given this prima facie evidence for the benefits of programming, we felt it worthwhile to tackle the transfer issue directly.

The study reported in this paper focused on trying to obtain evidence that procedural skills do transfer from programming to other domains. We examined the issue by testing students on a set of algebra word problems. The kind of problems we used were based on previous research, which will be reviewed in the next section. This research identified a set of algebra word problems that elicited errors that could be traced to the adoption of a descriptive rather than a procedural approach to solving the problem. Our intent is to examine whether programming, by

encouraging students to be more procedural in their problem-solving, can improve performance on these algebra word problems.

2. PREVIOUS RESEARCH

2.1. Algebra Word Problems

In some recent research, Clement and his colleagues [2]found that students seemed to have a lot of difficulty in translating simple algebra word problems from the description of the problem in English into an algebraic equation. Two typical problems are shown in Table 1. Among college freshman engineering students, 37% missed the first problem while 73% missed the second. In their experiments, Clement and his colleagues were able to eliminate difficulty with algebraic manipulation and tricky wording as major sources for the errors. The errors made on problems 1 and 2 were largely of one kind; most were *reversals*: 6S=P instead of S=6P and 4C=5S instead of 5C=4S. The consistency of these error patterns across these and other problems, and analysis of thinking aloud protocols, argues against the idea that they were caused by carelessness, and suggests that they stem from conceptual bugs.

Clement and his colleagues carried out a number of videotaped interviews in order to understand the source of these errors [2]. Using this technique they were able to identify a number of strategies which led to the reversal error. What seemed to underlie the strategies, was that students seemed to be adopting a descriptive approach to the problem. The incorrect equation and the descriptive approach contrasts with the correct equation S = 6P, which needs to be viewed as expressing an active operation being performed on one number (the number of professors) in order to obtain another number (the number of students).

The results of these studies lend themselves to the following analysis:

- Students are making errors on algebra word problems because they are adopting a descriptive rather than a procedural approach.
- Programming provides an environment in which students are encouraged to develop a procedural view of problem solving [8]. In particular, we found when programmers were asked to provide a computer program as a solution to "students & professors" type problems, they were more often correct than when asked to simply provide an algebraic equation. Note carefully that the key line in the computer program is precisely the same algebraic equation that would be required for a correct solution to the algebra word problem!

These claims lead to the following prediction: If the kind of skills we have identified as being associated with programming do transfer to other domains we should find that students do better

PROBLEM 1

Given the following statement:

"There are six times as many students as professors at this University".

Write an equation to represent the above statement. Use S for the number of students and P for the number of professors.

- Result: 63% correct
- Typical wrong answer: 6S = P

PROBLEM 2

Given the following statement:

"At Mindy's restaurant, for every four people who order cheesecake, there are five people who order strudel."

Write an equation to represent the above statement. Use C for the number of cheesecakes ordered and S for the number of strudels ordered.

- Result: 27% correct
- Typical wrong answer: 4C = 5S

Table 1:
EXAMPLES OF ALGEBRA WORD PROBLEMS

on algebra word problems after completing a programming course as compared with their performance before they have taken the course.

2.2. Pilot Study

Overall. In a preliminary study [4] we compared the performance of a group of 31 students who had just completed an introductory programming course with a similar group of 26 students who had no programming experience. We gave both groups of students a test which included a number of algebra word problems of the kind shown in Table 1. The results were consistent with our predictions; the students who took the programming course performed better than the students who had no exposure to programming. The programmers averaged 69% correct whereas the non-programmers averaged only 54% correct (ANOVA: $F_{1, 53} = 4.40$, $p < 0.05$).

Problem Decomposition. We also obtained results which pointed to specific effects related to programming. In the earlier study on algebra word prob'ems, Clement et al. [2] found that students performed more accurately on problems in which only one of the variables was multipled by a number (e.g., S = 6P) as compared with problems in which both variables were multiplied by a number (e.g., 5C =4S). Both of these problems were included in the preliminary study. For the sake of brevity we refer to the first, simpler problems as an *integral* problem, and the second problem as a *non-integral* problem. The data from the two problems are shown in Table 2. These data indicate that although the nonprogrammers are still performing worse on the non-integral than on the integral problems, the programmers are performing equally well on both. One explanation for the better performance by the programmers is that they developed skill in problem decomposition and that they were applying that skill to the non-integral problems.

Why should programming help problem decomposition? One answer is that students are taught to solve problems by breaking them down into small manageable components. For instance, we have observed students who solved a non-integral problem by writing the equation as

```
X = C * 5
S = X / 4
```

instead of writing the equation, S = 5/4 * C directly. The skill that is exercised here is both decomposition and composition. The student must be able to see that a single equation can be broken down into two parts, and that a correct solution can be composed by using an additional variable to connect the separate parts. The advantage of decomposing a problem in this way, whether it is done explicitly or not, is that it is often easier to generate two simple equations than a single, more complex one.

Equations as active operations. The second result which demonstrated influence of the programming course occurred on a set of problems which were constructed to elicit improvements

		NON-PROGRAMMERS	PROGRAMMERS
	Number of subjects	(N = 26)	(N = 31)
GENERATE			
	Correct Answer	%Correct	%Correct
NON-INTEGRAL			
At Mindy's restaurant for every 4 people who ordered cheesecake, there were people who ordered strudel.	4S=5C	39%	67%
INTEGRAL			
At a Yankees game for every 3 hot dog sellers there is a Coke seller.	H=3C	62.5%	68%

TASK: Write an equation to represent the statement

Table 2: PERCENT CORRECT FOR INTEGRAL AND NON-INTEGRAL VERSIONS OF EQUATION GENERATION PROBLEMS: PILOT STUDY

in performance. Equations for the word problems given in Table 1 can be written equivalently as a *ratio*: C/S = 4/5; as a *multiple* expression: 5C = 4S; or with a *single variable* on one side of the equation: C = 4/5 S. In an unpublished study, we found that there was a strong correlation between the form in which students wrote their equation and the accuracy of that equation. Equations written as multiples were commonly incorrect (i.e., 4C = 5S), while equations written in either of the other two forms were more commonly written correctly. Indeed, the previous research on algebra word problems noted that many of the incorrect equations were associated with a simple word-order match strategy in which the order of numbers and variables in the equation matches the order these were mentioned in the problem description. This strategy, of course, will generate an incorrect multiple equation, 4C = 5S. We reasoned that if students were making errors because of this strategy, showing them an equation not written as a multiple might elicit better performance. To test the effect of the form of the equation on performance, we gave students partial equations to complete; examples of the problems and the results are shown in Table 3.

Two important results emerged. Firstly, we found that students were more accurately on equations that were written in the form of a ratio than on equations written as a multiple. This result is perhaps not surprising given that the problems we were using are properly classified as ratio problems. The result is more important, however, for its demonstration that the completion task can elicit more accurate performance, and, by implication, that the task is sensitive to changes that might result from a programming course. The effect of a programming course on algebra word problems can be seen in the next result.

Secondly, we found that the programmers performed better than the non-programmers on problems in which the equation was written with a single variable on one side. This is the form in which equations and assignment statements are most often written in a program. Moreover, assignment statements in a program convey the notion of an active operation in which the output variable (e.g., S in the assignment statement, S := 5/4 * C) is assigned the result of the operation on the input variable (e.g.,the result of 5/4 * C). One interpretation of this result is that programmers are exposed not only to a particular form of equation, but that they are exposed to the notion of an equation (as an assignment statement) being associated with an active operation. This interpretation amounts to the claim that programming helps students overcome their descriptive approach to equations and instead encourages them to take a more active, procedural approach.

	Number of subjects		NON-PROGRAMMERS (N = 26)	PROGRAMMERS (N = 31)
	Equation given	Correct Answer	%Correct	%Correct
COMPLETE THE EQUATION				
A: SINGLE VARIABLE FORM OF EQUATION When the slot machine at Rosie's bar gives a jackpot, there are 6 nickels for every 5 quarters.	$N = \frac{?}{?} Q$	$N = \frac{6}{5} Q$	35%	61%
B: RATIO FORM OF EQUATION In Fairmont Hills there are 8 plumbers for every 3 electricians.	$\frac{?}{?} = \frac{P}{E}$	$\frac{8}{3} = \frac{P}{E}$	65%	77%
C: MULTIPLE FORM OF EQUATION When Elizabeth Taylor goes to Tiffanys she buys 3 rubies for every 2 emeralds.	?R = ?E	2R = 3E	35%	48%

TASK: Complete the given equation by replacing the question marks

Table 3: PERCENT CORRECT RESPONSES FOR THE EQUATION COMPLETION PROBLEMS: PILOT STUDY

BEST COPY AVAILABLE

2.3. Summary of Previous Research

Previous research suggests that there is a class of problems, algebra word problems, in which students make errors because they adopt a descriptive rather than a procedural approach to the problem. Programming, as we argued earlier, encourages students to develop a more active, procedural approach to problem solving. Students who have taken a programming course should be able to transfer the skills they have learned in that course to other problems, in just those cases where a procedural-based approach is needed. Initial support for this claim is encouraging. In the pilot study described above, we reported some evidence to suggest that two aspects of problem-solving - problem complexity and an active approach to equations, did seem to be improved by programming experience.

3. CURRENT STUDY

The present study was designed to follow up the preliminary results from the pilot study in a more experimentally controlled setting. The study focused on the following question:

1. Do students improve their performance on algebra word problems as a result of taking a programming course?
2. Can we attribute any improvement in performance to the development of particular skills or strategies?
    - skills associated with problem decomposition
    - strategies associated with viewing an equation as an active operation
    - generality of transfer across problem types

3.1. Materials

The first question was addressed by comparing performance for each student over the whole test. The test consisted of 36 items; these items are described below.

The second question has three parts: ***Problem Decomposition***, ***Problem Strategy***, ***Problem Type***.

Problem Decomposition. We examined skills associated with problem decomposition by varying the complexity of the problem. In addition to the *integral* and *non-integral* versions used in the pilot study, we added a third level of complexity, called *combination* problems. In these problems, both an integral and a non-integral equation are required to solve the problem. Examples of the problems are given in Table 4. If programming teaches problem decomposition and students can transfer that skill, we should find that students from the programming course show more improvement on the complex problems compared with the simple problems.

Problem Strategy. We examined whether students will view equations in a more active way after taking a programming course. To examine this question we used the same completion task

PROBLEM 1: INTEGRAL

Write an equation (or equations) to represent the following statement:

"At the bookstore, for every 9 copies of the Times there is a copy of the Gazette."

PROBLEM 2: NON-INTEGRAL

Write an equation (or equations) to represent the following statement:

"At Mindy's restaurant for every 4 people who ordered cheesecake there were 5 people who ordered strudel."

PROBLEM 3: COMBINATION

Write an equation (or equations) to represent the following statement:

"The candy store sells 4 bars of chocolate for every 3 ice-creams it sells and it also sells 5 ties as many bars of chocolate as candies."

Table 4:
EXAMPLES OF RATIO PROBLEMS AT THREE LEVELS OF COMPLEXITY

we used in the pilot study. In this task, students are given a partial equation to complete instead of generating an entire equation. The test included 12 of these completion problems. As in the previous study, the three alternate forms of writing an equation were used. These forms are: ratio, single variable and multiple. Each form occurred equally often (i.e., 4 times) across the total of 12 problems. Examples of the problems are shown in Table 5. If students approach problems in a more active manner we should find that this change is picked up by their response to the *single variable* forms of equations because this form of equation is most similar to an assignment statement in a program. Moreover, if programming encourages a different approach to problem solving rather than simply exposing students to a particular form, we should find improvement on both the single variable forms and on the multiple forms.

Problem Type. In the pilot study we used only one kind of algebra word problem; those involving ratios. In order to broaden the scope of our inquiry into transfer effects, the present study included a set of algebra word problems that involved percentages. Percentages are commonly used in algebra word problems, for example, to calculate interest or to assess the amount of different ingredients in a mixture e.g., [5]. Percent problems are of particular interest here because they can be constructed to have the same surface form as the ratio problems, even though their underlying structure and method of solution may be different. An example is:

> Write an equation (or equations) to represent the following statement:
>
> "The number of paper bags on the ground at the park is 45% of the number of people visiting the park that day."

The correct equation could be written:

PB = 45% V

where PB represents the number of paper bags and V represents the number of visitors in the park. We varied the complexity of the percent problems in the same way as for the ratio problems. Examples of the problems are shown in Table 6.

There are a number of superficial similarities between the percent and ratio problems. However, there is one very important difference. In the percent problems, a word order match strategy (i.e., a non-procedural approach) can be used to generate a *correct* equation; this strategy will lead to an incorrect equation for the ratio problems. If programming gives students a general improvement in performance, we should find that students improve on both problems as a result of taking a programming course. Alternatively, if programming is associated with teaching procedural skills then there should be more improvement for the ratio problems since these problems benefit more then the percent problems from a procedural approach.

RATIO FORM

Given the following statement:

"In Fairmont Hills, there are 6 plumbers for every 5 electricians."

Let P represent the number of plumbers and let E represent the number of electricians. Complete the equation given below by replacing the question marks.

$$\frac{?\ E}{?\ P}$$

SINGLE LETTTER FORM

Given the following statement:

"At the last company cocktail party there were 3 people who drank wine for every 7 people who drank beer."

Let W represent the number of wine drinkers and let B represent the number of beer drinkers. Complete the equation given below by replacing the question marks.

$$B = \frac{?}{?} W$$

MULTIPLE FORM

Given the following statement:

"When Elizabeth Taylor goes to Tiffanys she buys 6 rubies for every 5 emeralds."

Let B represent the number of rubies and let E represent the number of emeralds. Complete the equation given below by replacing the question marks.

$$?E = ?B$$

Table 5:
EXAMPLES OF PROBLEMS USED IN COMPLETION TASK

BEST COPY AVAILABLE

PROBLEM 1: INTEGRAL

Write an equation (or equations) to represent the following statement:

"The number of paper bags on the ground at the park is 45% of the number of people visiting the park that day."

PROBLEM 2: NON-INTEGRAL

Write an equation (or equations) to represent the following statement:

"70% of the number of long distance calls that Mary made accounted for 30% of the total number of calls she made."

PROBLEM 3: COMBINATION

Write an equation (or equations) to represent the following statement:

"20% of the students at a high school own cars.
40% of the students with cars make up 80% of the students with jobs."

Table 6:
EXAMPLES OF PERCENT PROBLEMS AT THREE LEVELS OF COMPLEXITY

3.2. Design

The test consisted of 12 completion problems and 24 problems which required students to either generate an equation or a numerical solution when supplied with the value of one of the variables. For the 24 "generate" problems 2 factors were independently varied: problem complexity (integral, non-integral, combination) and problem type (ratio, percent). Crossing these 2 factors yielded 3 x 2, i.e., 6 experimental conditions. There were 4 problems/condition, giving a total of 24 test items. The main measure of performance was accuracy.

Two tests were constructed each with 36 items; one called Test A and the other called Test B. The two tests differed only in the particular lexical items that were used in the problems; in all other respects the tests were identical. Subjects were given one test at the beginning of the semester (e.g.,Test A) and the other test at the end of the semester (e.g.,Test B). Subjects were randomly assigned either Test A or Test B for the first testing session.

3.3. Procedure

The study was run using the following procedure. Subjects were college students who were enrolled in an introductory programming course. The language they were taught was Pascal. A complete test of the 36 algebra word problems was administered to each student at the beginning of the semester, before they had taken the programming course, and again at the end of the semester after they have finished the course. In order to be confident that the results we obtain are due to the programming course rather than to other factors involved in a test-retest situation, we also gave the same test to a group of students who were not enrolled in a programming course. This second 'control' group were carefully selected so that their background matched that of the experimental group as closely as possible. In particular we asked all students to allow the university to release to us their SAT scores so that we had some independent assessment of their general ability level.

3.4. Subjects

Experimental Subjects. These subjects were recruited from an introductory programming course in the computer science department at a large state university. The course we recruited in was open to students throughout the university and was taught at a level appropriate for students who were not majoring in computer science. The department did offer a parallel course that semester for computer science majors. Subjects were paid $5 for participating in the first pre-test and an additional $15 if they returned to take part in the second post-test.

A total of 132 subjects participated in the initial pre-test. 92 of these subjects, i.e., 70%, returned for the post-test session at the end of the semester. Because the study was designed to test the effect of programming on performance on algebra word problems, we eliminated those

subjects who had taken previous programming courses at college level from further consideration. By this criterion we rejected 10 out of the 92 subjects who had completed both the pre-test and the post-test. We were thus left with a core group of 82 subjects who had completed both the pre-test and the post-test and who had no previous significant programming experience.[1]

Control Subjects. This group of subjects were recruited from upper level psychology courses, specifically a course on motivation and a cognitive psychology course. Subjects were given experimental credit for participating in the first, pre-test and experimental credit + $10 for returning to take the second, post-test.

A total of 46 subjects participated in the initial pre-test. 2 of these subjects turned out to be enrolled in the programming course and have been included in the data for the experimental subjects. Thus, there were 44 control subjects who took the initial pre-test. 32 of these subjects, i.e., 73%, returned for the post-test session at the end of the semester. From this initial pool of 32, we eliminated those subjects who had significant prior programming experience, using the same criterion as we used for the experimental subjects. By these criteria, we eliminated 4 subjects. We were thus left with a core group of 28 subjects who had completed both the pre-test and the post-test and who had no previous significant programming experience and who were not currently enrolled in a programming course.

In terms of background, 79% of the programmers had arts majors with the remaining 21% having majors in science or engineering. All the non-programmers had non science majors. Thus most of our subjects, both experimental and control, were not science majors.

4. RESULTS

Before going through the results we need to point out that after carrying out the study, we found that we had an uneven distribution of men and women in our two groups of subjects. The genderes were evenly distributed for the programmers: there were 42 women and 40 men. However, of our core group of 28 control subjects, 24 of them were women and 4 of them were men. If it were the case that men and women responded the same way across all conditions, we would be justified in ignoring gender and simply evaluating performance on the basis of the pre-test and the post-test for the two groups of subjects. However, as will become apparent, we found differences in performance between men and women throughout all the data analyses. Thus, it must be stressed, that although we will present the data for all groups of subjects, the data from the male control group are not reliable because the sample size is too small, and the

[1]We did not exclude subjects who said they had taken a programming course such as BASIC in high school, since our previous research has led us to believe that these courses do not exert sufficient influence on a student's problem solving to warrant the exclusion of that student.

variances in performance too large to allow us to perform meaningful statistical comparisons. We have, however, included the data in the tables to give some sense of how these 4 people performed.

4.1. Overall

The first set of results we will present relate to our first question:

- Do students improve their performance on algebra word problems as a result of taking a programming course?

The data which are presented in Table 7 indicate that any support for that question has to be qualified. In particular, the data indicate that the men did improve as a result of taking the programming course but the women did not. An analysis of the data confirm that observation; when level of improvement was compared, the men programmers showed more improvement than the women (ANOVA: $F_{1,\ 80} = 7.4$, $p < 0.01$). The men showed a highly significant improvement in performance when tested separately from the women, (t test: $t_{39} = 4.96$, $p < 0.0005$). It must be stressed, however, that the data we are discussing is *improvement* in performance. The women are performing as well, in fact better, than the men on the initial test. That is, the data imply that men and women are equally capable of solving algebra word problems, but, for some reason, men are more likely to improve their performance.

We need to add a further qualification to our result. In the absence of an adequate control group of men non-programmers, we cannot draw a definitive conclusion about the contribution of the programming course to the improvement in performance. Indeed, the data in Table 7 show that there was some improvement for the men non-programmers, although that improvement was not reliable (t test: $t_3 = 1.66$, n.s.) because of the small sample and the extremely high variance in their data.

The data show that the women programmers did not improve their performance on the test. One might want to conclude from those data that the women programmers, did not transfer their skills and learning from programming to the algebra word problems. An alternative explanation is that the women have higher SAT scores on average than the men (see Table 7). Thus there may be more women than men who are already performing near their optimum level on the pre-test and hence who are not able to improve their performance. This alternative explanation was not borne out by our analysis. We found that SAT scores accounted for only .1% of the variance in the improvement scores, while gender accounted for 9%. That is, a subject's SAT score was not a very good predictor of improvement, but the gender of the subject did predict improvement.[2] Thus, their higher SAT scores do not account for the failure of the women to

[2]We also looked at the data for the individual subjects and found that 17 of the 42 women programmers actually did worse on the post-test than on the pre-test. Only 7 of the 40 men showed a decrement in performance.

ALL PROBLEMS

	PROGRAMMERS		NON-PROGRAMMERS	
	MALE (N = 40)	FEMALE (N = 42)	MALE (N = 4)	FEMALE (N = 24)
SAT	520	560	540	500
PRE-TEST	48% (4%)	55% (4%)	54% (12%)	44% (6%)
POST-TEST	61% (4%)	57% (4%)	66% (16%)	45% (5%)
DIFF	13%	2%	12%	1%

Table 7: Percent correct for problems overall.
The total number of problems was 36.
The standard errors are given in parentheses.

improve.

We also found that, in general, the men were more consistent than the women, particularly with respect to the relation between performance in the programming course and amount of improvement. It might be expected that students with higher grades in the programming course should show more improvement because they have a better understanding of programming to transfer (assuming that the grades are measuring programming understanding). This expectation was borne out for the men; those with higher grades in the course showed more improvement than those with lower grades (correlation = 0.36, $p < 0.05$). There was, however, no correlation between course grade and improvement, for the women (correlation = 0.01). It should be noted that the women received the same grades in the course, on average, as the men.

In Table 8, we show the pre-test and post-test scores for the men and women as a function of grade. The mean SAT score for the subjects at each grade level is also listed. The data for the men show a highly consistent and regular pattern of performance across grade, SAT, pre-test score, post-test score and difference (i.e., improvement) score. The pattern for the women is much less consistent.

What can we conclude from these data? One conclusion certainly seems to be that our test is not eliciting the same evidence of transfer from the women programmers as from the men. There are a number of requirements for programming skills to transfer. One is that the student acquire a good enough understanding of programming to be able to change previous problem-solving behavior. A second is that the student perceive, either consciously or unconsciously, that there is some commonality between the algebra word problem and programming. If a student keeps knowledge of each subject area locked in separate mental compartments, there is no communication for transfer to take place. A third requirement is that sufficient time has to elapse for well-rehearsed behaviors such as those used in the algebra word problems, to be replaced by the newer behaviors learned in the programming course. The women programmers may have a different agenda than the men with respect to any of these requirements.

The differences between the men and women could of course, reflect some idiosyncracies of the particular students we tested, and certainly further studies need to be conducted to verify our findings. However, to the extent that the present results are valid they offer some intriguing differences between men and women that only show up on certain types of problems and when students are tested for changes in performance.

MALES

GRADE	Mean SAT	N=	PRE-TEST	POST-TEST	DIFF
TOP	545	12	21.8	27.6	5.8
HIGH	525	9	17.7	24.8	7.1
MEDIUM	501	12	15.3	18.5	3.2
LOW	507	6	11.8	11.2	-0.6
unknown		1			

FEMALES

GRADE	Mean SAT	N=	PRE-TEST	POST-TEST	DIFF
TOP	587	16	25.4	26.2	0.8
HIGH	539	8	15.5	14.4	-1.1
MEDIUM	565	13	17.8	19.6	1.8
LOW	457	4	13.7	9.4	-4.2
unknown		1			

Table 8:
The mean pre-test and post-test scores for male and female programmers as a function of grade in programming course

4.2. Particular skills and strategies

The second question we posed was:

- Can we attribute any improvement in performance to the development of particular skills or strategies?

We proposed to examine this question in the context of problem decomposition, problem strategy and problem type.

4.2.1. Problem Decomposition

The ratio and percent problems were presented at three levels of complexity - integral, non-integral and combination kinds of equations. The data for these problems are shown in Table 9. The men programmers show the predicted pattern of performance; the more complex the problem the more they improve. However, we also found some improvement for the men non-programmers. Specifically, this group showed a similar pattern of improvement to the men programmers. Moreover, for the non-integral problems the improvement was significant (t test: $t_3 = 4.9$, $p < 0.02$). Thus, we cannot conclude that it is just programming that helps people to cope with complexity; the men non-programmers had no programming experience and were still able to improve. However, as we have pointed out before, the data from this group does not provide a reliable indication of the performance of men who have no programming experience. We thus must conclude by saying that programming may have some impact on how well people cope with complexity, but further studies need to be done to tease out just how much programming helps.

4.2.2. Problem Type

The test included 12 problems involving ratios and a matching set of 12 problems that involved percentages. The ratio and percent problems were constructed to match as closely as possible in all ways except for whether the problem was one that involved percentages or one that involved ratios. However, one of the important differences between these two types of problem is that a procedural approach benefits the ratio problems more than the percent problems.

When we looked at the data for these two problem types we first found a dramatic difference in performance between men and women; the data are shown in Table 10 and Table 11. On the ratio problems the men programmers improved (by 15%) while the women programmers showed no improvement. However, on the percent problems the women improved (by 9%) but the men did not. An ANOVA confirmed the three way interaction between gender (male/female) problem type (ratio/percent) and test time (pre/post) (ANOVA: $F_{1, 80} = 57.11$, $P < 0.002$). Note that the women programmers showed more improvement than the men on the percent problems even though their initial performance was higher than the men. These data further highlight that men and women differ in their style, rather than their accuracy of solving problems.

COMPLEXITY

PROGRAMMERS

	MALE (N = 40)			FEMALE (N = 42)		
	INTEGRAL	NON-INTEGRAL	COMBINATION	INTEGRAL	NON-INTEGRAL	COMBINATION
PRE	69% (3%)	50% (4%)	34% (5%)	69% (4%)	63% (4%)	40% (4%)
POST	69% (5%)	61% (4%)	47% (5%)	69% (4%)	61% (5%)	47% (5%)
DIFF	0%	11%	13%	0%	-2%	7%

NON-PROGRAMMERS

	MALE (N = 4)			FEMALE (N = 24)		
	INTEGRAL	NON-INTEGRAL	COMBINATION	INTEGRAL	NON-INTEGRAL	COMBINATION
PRE	69% (6%)	53% (9%)	38% (20%)	54% (6%)	47% (7%)	30% (6%)
POST	72% (16%)	78% (11%)	63% (14%)	58% (5%)	50% (6%)	31% (7%)
DIFF	3%	25%	25%	4%	3%	1%

Table 9: Percent correct as a function of the complexity of the problem. There were a total of 8 problems for each level of complexity. The standard errors are given in parentheses.

The improvement in performance by the women on the percent problems cannot be attributed to their programming experience, however. A comparison of the level of improvement between the women programmers and the women non-programmers revealed that there was no difference between them (ANOVA: $F_{1, 84} = 0.133$); the non-programmers improved as much as the programmers. The men programmers showed no improvement in performance. These data suggest that programming makes little or no contribution to performance on percent problems.

Both the men programmers and non-programmers showed some improvement on the ratio problems. The improvement for the men non-programmers was not statistically reliable (t test: $t_3 = 1.36$, n.s.) due to the high variance in the data.

The comparison of ratio and percent problems is consistent with the claim that programming helps students develop a procedural approach to problem-solving. This conclusion follows from the argument that a procedural approach should improve performance on the ratio problems only, which is what we found, albeit for the men programmers only, and with some question about the contribution of programming rather than gender.

4.2.3. Problem Strategy

Both the ratio and percent problems discussed above were presented in the context of a generate task; students were given a statement and asked to generate an equation to represent that statement. In another set of ratio problems we prompted students for the correct equation by giving them a partial equation and asking them to complete it. The equation was variously written in the form of a ratio, with a single variable on one side of the equation, or in a multiple form where each variable is multiplied by a single number. The data for these 12 completion problems are shown in Table 12. These data show dramatic improvement in performance for the men programmers (50%) (t test: $t_{39} = 4.95$, $p < 0.0005$). There was no improvement in performance at all for any of the other groups of subjects. In Table 13 we also show the data broken down by the form of the solution. Here it can be seen that the men programmers improved most on the multiple and single variable forms of the equations. Again, none of the other groups showed any substantial improvement in performance.

These data provide the strongest evidence so far for transfer effects from programming. Even though we cannot rule out other explanations, at least some of the improvement on these problems represents transfer from programming. It is also interesting to note that the men improved more on this completion task than on the set of ratio problems presented earlier which used a generate task. The difference between these two results was not reliable (t test: $t_{39} = 1.62$).

The completion problems, particularly those problems in which equations are presented in the form of a single variable on one side of the equation, represent the situation of nearest transfer in

PERCENT PROBLEMS

	PROGRAMMERS		NON-PROGRAMMERS	
	MALE (N=40)	FEMALE (N=42)	MALE (N=4)	FEMALE (N = 24)
	Correct	Correct	Correct	Correct
PRE	60% (5%)	61.5% (5%)	77% (12%)	46% (7%)
POST	61% (5%)	70% (4%)	94% (2%)	57% (7%)
DIFF	1%	8.5%	17%	11%

Table 11: Percent correct performance for PERCENT problems. There were a total of 12 problems in the set. The standard errors are given in parentheses.

RATIO PROBLEMS

	PROGRAMMERS		NON-PROGRAMMERS	
	MALE (N=40)	FEMALE (N=42)	MALE (N=4)	FEMALE (N = 24)
	Correct	Correct	Correct	Correct
PRE	43% (4%)	53% (5%)	31% (16%)	42% (6%)
POST	58% (6%)	48% (6%)	48% (24%)	36% (7%)
DIFF	15%	-5%	17%	-6%

Table 10: Percent correct performance for RATIO problems. There were a total of 12 problems in the set. The standard errors are given in parentheses.

COMPLETION TASK - FORM OF EQUATION

PROGRAMMERS

	MALE (N = 40)			FEMALE (N = 42)		
	RATIO	SL	MULTIPLE	RATIO	SL	MULTIPLE
PRE	63% (6%)	40% (5%)	26% (6%)	71% (6%)	46% (6%)	38% (6%)
POST	77% (5%)	61% (6%)	54% (6%)	66% (6%)	49% (6%)	42% (6%)
DIFF	14%	21%	28%	-5%	3%	4%

NON-PROGRAMMERS

	MALE (N = 4)			FEMALE (N = 24)		
	RATIO	SL	MULTIPLE	RATIO	SL	MULTIPLE
PRE	75% (18%)	63% (16%)	25% (14%)	65% (8%)	34% (7%)	33% (8%)
POST	81% (19%)	38% (24%)	50% (29%)	65% (8%)	34% (7%)	27% (8%)
DIFF	6%	-25%	25%	0%	0%	-6%

Table 13: Percent correct performance for the COMPLETION task.
There were a total of 4 problems for each form.
The standard errors are given in parentheses.

COMPLETION TASK

	PROGRAMMERS		NON-PROGRAMMERS	
	MALE (N=40)	FEMALE (N=42)	MALE (N=4)	FEMALE (N = 24)
	Correct	Correct	Correct	Correct
PRE	43% (4%)	52% (5%)	54% (14%)	44% (6%)
POST	64% (5%)	52% (5%)	56% (21%)	42% (6%)
DIFF	21%	0%	2%	-2%

Table 12: Percent correct performance for the COMPLETION task.
There were a total of 12 problems in the set.
The standard errors are given in parentheses.

this study. It is thus not surprising that we should find the greatest transfer where there is the greatest similarity between the items in the test and the kind of problems the student gets in the programming course. However, more subtle skills must also be transferred from programming to algebra word problems to account for the exceptional improvement in performance on the problems which were presented in the multiple form. Students are not likely to see equations written in this form in computer programs. However, this form of writing an equation is the form that is most strongly associated with making errors. The students who improved their performance when the equation was presented in this form may very well have realized how to write correct equations in a single variable form. They may have then completed the multiple form equations by first changing it into the single variable form, solved it that way, and then changed it back into a multiple.

Our conclusion from these data is that they provide quite strong evidence not only that a programming course can improve performance, but that programming encourages students to develop skills associated with viewing equations as active procedures rather than as descriptions. Both the data from the comparison of the problem types and these data from the completion problems are consistent with that conclusion.

5. DISCUSSION

There is a strong belief that students who learn programming for the first time should be able to transfer some of their new programming skills to other disciplines. However, it has proven extremely difficult to provide empirical evidence for this belief in part because there is a lot of uncertainty over where to look for transfer effects.

In the present study we took a very focused approach to the transfer issue by examining whether a programming course improved procedural skills. In particular we examined whether programming helps students decompose problems, whether it helps them develop a more active approach to problem-solving and whether the kind of transfer we were examining was specific to particular non-procedural strategies. To further demonstrate the influence of programming on individual students, we tested the same students before and after they had taken their first, introductory programming course, on a set of algebra word problems. The performance of these programmers was compared with a similar group of students who had not taken a programming course.

Our data were inconclusive with respect to the transfer issue. Unexpectedly, we found a large gender difference which translated into the suggestion that men were far more likely than women to show transfer effects. That is, more of the men improved their performance on the set of algebra problems than did the women. Moreover, due to an inadequate control group for the men programmers, it was not possible to demonstrate whether this improvement was due to

gender differences or to programming experience. However, even if we take a pessimistic view of the data and treat the means from the control group of men as being representative of men non-programmers in general, there is some evidence for one of the procedural skills; the men programmers did adopt a more active approach to problem-solving after taking the programming course as compared with their previous performance. The improvement was most clearly demonstrated when the problem prompted the students to take a more active approach rather than when this approach was expected to emerge spontaneously.

It seems unlikely that students who take a new course in any subject do not learn something new that can be applied to other old, more familiar subjects. Despite the apparent strong face validity of transfer effects from programming, it seems very difficult to provide strong empirical evidence in its favor. Based on what we found in this study, future studies of transfer effects will need to pay more attention to individual differences, in addition to focusing on the specific types of skills that are learned in programming that might be transferred.

References

[1] Bonar, J., Ehrlich, K., Soloway, E.
Collecting and Analyzing On-Line Protocols from Novice Programmers
Behavioral Research Methods and Instrumentation 14:203-209, 1982.

[2] Clement, J., Lochhead, J., and Monk, G.
Translation difficulties in learning mathematics.
American Mathematical Monthly 88:26-40, 1981.

[3] Ehrlich, K., Soloway, E.
An Empirical Investigation of the Tacit Plan Knowledge in Programming.
Technical Report 82-30, Dept. of Computer Science, Yale University, 1982.

[4] Ehrlich, K., Soloway, E., Abbott, V.
Transfer Effects From Programming To Algebra Word Problems: A Preliminary Study.
Technical Report 257, Dept. of Computer Science, Yale University, 1983.

[5] Hinsley, D. A.,Hayes, J. R., and Simon, H. A.
From words to equations: Meaning and representation in algebra word problems.
Erlbaum, Hillsdale, NJ, 1977, :

[6] Howe, J.A.M., O'Shea, T. and Plane, J.
Teaching Mathematics Through Logo Programming.
Technical Report 115, University of Edinburgh, Artificial Intelligence, 1979.

[7] Papert, S.
Mindstorms, Children, Computers and Powerful Ideas.
Basic Books, 1980.

[8] Soloway, E., Lochhead, J., Clement, J.
Does Computer Programming Enhance Problem Solving Ability? Some Positive Evidence on Algebra Word Problems.
R. Seidel, R. Anderson, B. Hunter (Eds.), Academic Press, New York, NY, 1982b, pages 171-215.

[9] Soloway, E., Bonar, J., Woolf, B., Barth, P., Rubin, E., and Ehrlich, K.
Cognition and programming: Why Your Students Write Those Crazy Programs.
In *Proceedings of the National Educational Computing Conference.* NECC, No. Denton, Tx., 1981.

[10] Soloway, E., Ehrlich, K., Bonar, J., Greenspan, J.
What Do Novices Know About Programming?
Ablex, Inc., 1982, .

WHAT WILL IT TAKE TO LEARN THINKING SKILLS THROUGH COMPUTER PROGRAMMING?

Roy D. Pea

Center for Children and Technology
Bank Street College of Education

Imagine yourself as a visitor to a traditional farming society in West Africa. You have arrived as a cross-cultural psychologist to study whether and how literacy affects the way people think. Let us begin by asking why you are here.

The acquisition of literacy had long been claimed to promote the development of intellectual skills. Prominent historians and psychologists had long argued that written language has many important properties that distinguish it from oral language, and that the use of written language leads to the development of highly general thinking abilities, such as logical reasoning and abstract thinking. Piagetian studies in other cultures had made clear that the kind of abstract thinking associated with formal operations did not develop in oral cultures. By contrast, when one looked at cultures that used written language, various cognitive tasks revealed high logical competencies.

But you had observed that studies bearing on this claim had always been done in societies such as Senegal or Mexico, where literacy and schooling were confounded. Perhaps schooling was responsible for these changes in thinking, rather than the use of written language per se.

This essay also appeared in the Preproceedings of the National Logo Conference, MIT, Cambridge, MA, June 1984, under the title of "Symbol systems and thinking skills: Logo in context." I would like to take this opportunity to thank the Spencer Foundation and the National Institute of Education (Contract #400-83-0016) for supporting our research program on the development of Logo programming and its relation to other cognitive skills. My colleagues at the Center for Children and Technology have been a continuing source of encouragement and stimulation.

The reason you have travelled to Africa is that you plan to test, for the first time, the cognitive effects of literacy independently of schooling. The society you are studying--the Vai--does not transmit literacy in the Vai written language through formal schooling. Their reading and writing are practiced and learned only through the activities of daily life. The Vai invented their written language a mere 150 years ago, and have continued to pass literacy on to their children without schooling.

Like all the psychologists before you, you have brought along suitcases filled with psychological tests and materials for experiments on concept formation and verbal reasoning. Results from performances by the Vai with and without written language experience will tell you whether possessing literacy affects the way they think. You then carry out your research.

As you look over your results from several years of work, you find no general cognitive effects of being literate in the Vai script. For example, the literate Vai were no better than the nonliterate Vai in categorization skills or in syllogistic reasoning. Literacy per se did not appear to produce the general cognitive effects on higher thinking skills you expected.

So you mull over this fact for some time. How could this be? The arguments were so plausible for why written language would affect the way people think. You wonder -- could the studies be done more carefully?

Before continuing this research strategy, you realize that there is a radically different way to think about your project. When you arrived you took for granted that literacy would have its general effects, and then looked to see if it did so by testing for general intellectual benefits. But with several years of survey and ethnographic observations under your belt, you have come to better understand the tasks that Vai literates encounter in their everyday practices of literacy. How does this relate to your experiments?

What you decide you could do instead is to actually look to see how literacy is practiced in the Vai culture. What is done with the written language? And then you ask a very different type of research question: How could what the Vai people do specifically with written language affect their thought processes? You decide, in other words, to let your fieldwork on literacy practices dictate the design of your "outcome" tasks. You thereby gain a great deal of precision in your hypotheses for the cognitive effects of literacy.

This reorientation literally turns on its head your paradigm of looking for general cognitive effects of literacy. You have abandoned the approach of making general predictions from developmental theory to effects on general intelligence. You now start with concrete observations of literacy behavior and build up to a general functional theory of the cognitive effects of specific literacy practices.

With this new approach you find that the Vai use their written language primarily for letter-writing, and for recording lists and making technical farming plans.

You then begin a new phase of research, to tease out cognitive effects of specific literacy practices rather than literacy per se. You design new tasks for assessing literacy effects that draw on related skills to those required by the practices you observe, but which involve different materials.

What you find when guided by this new functional perspective are dramatic cognitive effects of literacy. But they are more restricted in nature. For example, letter writing, a common Vai literacy practice, requires more explicit rendering of meaning than that called for in face to face talk. So you refine a communication task where the rules of a novel board game must be explained to someone unfamiliar with it, either face to face or by dictating a letter for an absent person. You find, lo and behold, that performances of Vai literates are vastly superior on either version of this task to those of nonliterates.

This is no mere parable. It is an account of an extensive five-year research project carried out by Professors Sylvia Scribner and Michael Cole (1981). It is the account of an intellectual voyage not so far removed from what children are learning with Logo programming. We can fruitfully apply the schema of this Vai story to questions about the cognitive effects of programming. And here, I believe, is where one will find what will be needed to learn thinking skills through programming.

Here, too, there are persuasive and intuitively appealing arguments for why people should become better thinkers by virtue of the use of a powerful symbol system such as the Logo programming language. It is alleged that children will acquire general cognitive skills such as planning abilities, problem solving heuristics, and reflectiveness on the revisionary character of the problem solving process itself. The features of programming literacy assumed here include the necessarily explicit nature of writing program instructions, the strategic and planful approaches ingredient to modular program design, and experience with the logic of conditionals, flow of control, and with program debugging.

But for programming languages, unlike written natural languages, we do not have the benefit of known historical and cultural changes that appear to result in part from centuries of use of the written language. The symbol systems provided by programming languages are relatively new. They have certainly changed the world; we now live in an information age because of achievements made possible by these languages. But what does it mean for how individuals think and learn?

Let us move our West African story to the context of the American Classroom. Here again we enter as psychologists, looking for general cognitive effects, much like the first literacy questions of the African enterprise.

Of course we assume that we know what kind of a mind-altering substance programming is (having been so affected ourselves), and we assume that "programming intelligence" and the kinds of programming activities carried out by adults will affect children too.

But we should give pause--for we have entered another culture. What will children do with a programming language in a discovery-learning situation, in Logo's "learning without curriculum" pedagogy (Papert, 1980), without benefit of being shown what kinds of things can be done, or being taught about the powers of the system or of thinking skills?

Nonetheless, without benefit of such hindsight, what do our psychologists in the Logo classroom do? Here we refer to our own work. They, too, look for programming's "effects," guided by somewhat the same kind of thinking that possessed the first phase of the Vai studies. The primary difference was that instead of testing for increments in general intelligence, or concept formation, they thought they were looking at more specific effects, quite plausibly linked to programming activities. Planning skills were the central focus, not abstract reasoning, which is only indirectly related to programming.

The psychologists' reasoning went something like this: Both rational analyses of programming and observations of adult programmers show that planning is manifested in programming in important ways. Once a programming problem is formulated, the programmer often maps out a program plan or design that will then be written in programming code. Expert programmers spend a good deal of their time in planning program design, and have many planning strategies available, such as problem decomposition, modular documentation, subgoal generation, retrieval of known solutions, and evaluative analysis and debugging of program components (e.g. Pea & Kurland, 1983).

Our psychologists studying the cognitive effects of Logo created planning tasks to reveal the development of different planning strategies, and of skills at plan revisions analogous to program revisions. But in two different studies, after a year of Logo programming, no effects of programming on performances in these planning tasks were found (Pea & Kurland, 1984). Children improved with age and practice on the planning tasks, but non-programmers did just as well after a year's time as did Logo programmers. Once again, like the researchers in West Africa, we

must reflect on our first set of assumptions for framing the research questions, and reconsider the meaning of our research findings.

Let us take a different, functional or activity-based approach to programming. Consider "programming" not as a unitary given, whose features we know by virtue of how adults do it at its best, nor as what it looks like in its ideal text-book forms. Let us look at programming as a set of practices that emerge in a complex goal-directed cultural framework of thought, emotion, and action.

Viewed in this way, by analogy to the Vai studies on literacy practices, we see that programming is as various and complex an activity matrix as literacy. Just as one may use one's literacy in Vai society to make laundry lists rather than analyze and reflect on the logical structures of written arguments, so one may achieve much more modest activities in programming than dialectics concerning the processes of general problem solving, planning, precise thinking, debugging, and the discovery of powerful ideas. One may, in particular, write linear brute-force code for drawing in turtle graphics.

Stated baldly, from a functional perspective we may see that powerful ideas are no more attributes inherent "in" Logo than powerful ideas are inherent "in" written language. Each may be put to a broad range of purposes. What one does with Logo--or written language--or any symbol system, for that matter--is an open matter. One must come to these powerful ideas and potentially fertile grounds for developing general thinking skills through discovery, or through learning with the guidance of others. Independent discovery and practice of Logo recursion, for example, may be a very rare spontaneous occurrence. The Vai in Africa have not spontaneously got onto the logical features of written language, philosophy, and textual analysis that written language allows. Likewise, most of our students--from grade school up through high school--have not spontaneously got onto the programming practices, such as planning to reuse procedures as building blocks in other programs, use of conditional or recursive structures, or careful documentation and debugging, that Logo allows.

For the Vai society, one could imagine introducing new logical and analytic uses of their written language. Similarly, one could imagine introducing to children the Logo programming practices many educators have heretofore taken for granted will emerge. In either case, we would argue that without some functional significance to the activities for those who are learning the new practices, there is unlikely to be successful, transferrable learning. Serving some purpose--whether being able to solve problems one could not otherwise, satisfying an intrinsic interest in complex problem solving, or achieving solidarity with a peer group who define their identity in part by "doing" Logo or written language--is a necessary condition for the symbolic activities we are interested in promoting to be ones our learners find a commitment to.

It is my hunch that wherever we see children using Logo in the ways its designers hoped, and learning new thinking and problem solving skills, it is because someone has provided guidance, support, and ideas for how the language could be used. They will have pointed the way through examples, rules, and help in writing programs and discussing the powerful ideas. To call these rich activities "learning without curriculum" is misleading, and an overly narrow view of what constitutes curriculum, for any projected path toward greater competency that another person helps arrange can be thought of as a curriculum.

There are many profound consequences of this more general account of what is involved in thinking about Logo as potential vehicle for promoting thinking and problem solving skills. A functional approach to programming recognizes that we need to create a culture for Logo, in which students, peers and teachers talk about thinking skills, display them aloud for others to share and learn from, a culture that continually reveals how programming is a vehicle for learning general thinking skills, and that builds bridges to thinking about other domains of school and life. Such thinking skills, as played out in programming projects, would come to play functional roles in the lives of those in this culture. Dialog and inquiry about thinking and learning processes would become second nature, and the development of general problem solving skills so important in an information age would be a common achievement of

students. This vision could, in principle, be realized. I imagine that important cognitive effects of programming, or of literacy are possible, but only when certain uses of these symbol systems are practiced, not the ones most engaged in today. There is far too much faith today that Logo carries with it guarantees of cognitive outcomes, and there is already evidence that when these changes are not found, educators will be prematurely discouraged.

Where are we left after these two continents of travel? With reason for optimism. There are many streams of Logo activities and research that should go on, for plurality and diversity provide exciting grounds for emergent ideas. Communication among groups of students, teachers, psychologists, and computer scientists will help in the formation of a broad community exploring these issues. These streams will no doubt embody a diversity of assumptions about what will best help create the culture of Logo I have referred to, in which one will be more likely to find the cognitive effects on thinking skills so many take for granted. Similar Logo cultures may arise that center on math learning, or programming. Each is likely to require the construction of extensive microworlds for learning more specific than the Logo language itself.

It is uplifting that there are so many positive energies in education today. The enthusiasm for Logo as a vehicle of cognitive change is an exhilarating part of the new processes of education one can see emerging. Cultures with thinking tools like Logo can be created. But we must first recognize that we are visitors in a strange world--at the fringe of creating a culture of education that takes for granted the usefulness of the problem solving tools provided by computers, and the kind of thinking and learning skills that the domain of programming makes so amenable to using, refining, and talking about together.

References

Papert, S. (1980). Mindstorms. New York: Basic Books.

Pea, R. D., & Kurland, D. M. (1983). On the cognitive prerequisites of learning computer programming. Interim Report to the National Institute of Education (Contract #400-83-0016).

Pea, R. D., & Kurland, D. M. (1984). Logo programming and the development of planning skills. Technical Report No. 16. New York, NY: Center for Children and Technology, Bank Street College of Education.

Scribner, S., & Cole, M. (1981). The psychology of literacy. Cambridge, MA: Harvard University Press.

Making Programming Instruction Cognitively Demanding: An Intervention Study

John Dalbey
Francoise Tourniaire
Marcia C. Linn

Assessing the Cognitive Consequences of
Computer Environments for Learning (ACCCEL)
Lawrence Hall of Science
University of California
Berkeley, California 94720

This document has been produced under Contract No. JE 0400830017 from the National Institute of Education, U.S. Department of Education to the University of California and Far West Laboratory for Educational Research and Development. However, it does not necessarily reflect the views of the Institute, the Department, the University or the Laboratory.

BEST COPY AVAILABLE

Introduction

A plethora of recent reports on the status of education in the United States today call attention to the lack of higher cognitive outcomes from classroom instruction. The report of the National Commission on Excellence in Education, "A Nation at Risk," describes a pressing need for educational reform to create a "learning society." Similarly, the National Science Board, in a report entitled "Educating Americans for the 21st Century," has called for the "new basics," or the thinking skills required to cope with rapid technological and scientific advances. These and other reports emphasize the need for instruction which fosters problem solving and prepares students to deal with new technolcgical tools as they become available. Computer learning environments have the potential for imparting some of these important higher cognitive skills.

This paper reports the evaluation of instructional provisions designed to foster higher cognitive skill in a computer programming course. This intervention explicitly encourages novice programmers to engage in the problem-solving skill of planning.

Advantages of the Computer Environment

Several features of the computer learning environment can, if capitalized upon, increase the quantity and quality of cognitively demanding activities offered in schools. The

BEST COPY AVAILABLE

ACCCEL project has identified six features of this environment which make it capable of providing cognitively demanding activity. Three are characteristic of many school environments, and three are somewhat unique to computer learning.

The first feature common to both some school and some computer learning environments is complexity. Students can solve complex problems when using computers. For example, students can solve problems which require the management of large amounts of information, such as plotting graphs or computing compound interest. The second feature is challenge. The computer can challenge the student to solve problems such as figuring out the best move in a game or determining the most efficient path through a maze. The third feature is the provision for multiple solutions to problems. Students can write and compare several programs which do the same thing. These features are characteristic of some other school activities, such as writing reports or proving geometry theorems.

Three additional features of the computer learning environment are less characteristic of other classroom learning. First, the computer environment is interactive. The computer can respond quickly and informatively to the learner. Thus, students can try several approaches for reformulating their computer program and determine whether each of those approaches is successful. In contrast, it

BEST COPY AVAILABLE

often takes days or weeks for students to get responses to their homework assignments in other subjects. Second, the computer can provide precise feedback. Thus, the computer can tell exactly what happens when data is entered into a program. In contrast, when students write essays, they frequently get relatively imprecise feedback. For example, they may learn that their work merits a "B-". Third, computer learning environments are consistent. They give the same response when the same information is presented. Moreover, they provide the same response for all learners. In contrast, teachers do not necessarily respond identically to the same information, either because they are rushed or distracted, or because they are tailoring their response to the perceived needs of the student. When teachers behave as good tutors, their tailored responses to the students provide advantages not available in most computer learning environments. On the other hand, when teachers are distracted, their inconsistent responses can be less desirable than those characteristic of computer environments.

Programming Instruction

Currently, the most cognitively demanding activity readily available on the computer is programming. This situation is changing as new software becomes available. Eventually, software which demands higher cognitive skills, but is free from some of the drawbacks of programming may become preferable to programming for fostering higher cogni-

BEST COPY AVAILABLE

tive skill (Linn, 1984). Currently, however, pre-college students who have cognitively demanding interactions with computers are usually engaged in programming.

In spite of this situation, much programming instruction lacks a conceptual framework, and is not necessarily geared to fostering higher cognitive skills (Linn & Fisher, 1983). Until recently, most teachers did not have textbooks for programming instruction, but instead amassed materials somewhat haphazardly (e.g. Becker, 1984). Limited funds have been available for teacher professional development in computer education; many districts have followed a "buy hardware now, plan for its use later" approach. Before programming instruction can achieve reasonable goals, this situation must be rectified. The current study offers one approach.

Characteristics of Technological Experts

The contrast between the behavior of technological experts and the activities of students participating in pre-college programming classes motivated us to conduct this study. This discrepancy strongly suggests a need for materials which form a chain of cognitive accomplishments culminating in technological expertise. As matters now stand, some college teachers complain that pre-college instruction actually interferes with ability to profit from college computer education programs. Analysis of the nature of expertise, the characteristics of current pre-college

BEST COPY AVAILABLE

instruction and the potential of the computer learning environment, provides the primary rationale for this study.

The behavior of experts in computer learning environments contrasts sharply with the characteristics of current instruction as revealed by several studies of experts (Kurland & Pea, 1984; Linn, 1984; Jeffries, Turner, Polson, & Atwood, 1981). To create the chain of cognitive accomplishments which culminates in technological expertise students must learn to use the skills experts use everyday. Current instruction may not provide this opportunity.

One component of expertise is an extensive repertoire of "programming templates". Templates are stereotypic prescriptions for a particular aspect of a program, similar to schemata as described by Norman, Gentner, & Stevens (1976). Templates can apply to a whole program as exemplified in an "input-process-output" template. Templates can also apply to a specific function of a program such as a loop function as illustrated in Table 1. Research by Kurland & Pea (1984) reveals that both expert adult and expert student programmers can articulate their templates, recognize the relationships between their templates and new templates, and actively seek new templates.

Technological experts also use a variety of procedural skills. These are among the skills referred to as the new basics by recent commission reports. They are a part of the set of thinking or problem solving skills which individuals

BEST COPY AVAILABLE

need to survive in our society. An important component of this skill is planning or the ability to determine an appropriate sequence of available templates.

In the past, investigations of expert performance in formal systems such as solving mechanics problems in physics has proved informative for educators designing programs to foster these skills in novices (Larkin, McDermott, Simon & Simon, 1980; Heller & Reif, 1980). The current study was designed to examine the effects of making a novice programming course more cognitively demanding.

Literature on planning solutions to programming problems by experts indicates that experts engage in two complementary techniques: top-down design and stepwise refinement (Brooks, 1980; Atwood & Ramsey, 1978; Jeffries, Turner, Polson, & Atwood, 1981). Top-down design is an approach which decomposes a complex problem into subproblems. Experts can do this effectively, we surmise, because they have a large repertoire of program templates. Experts use their knowledge of templates to guide the decomposition process. Top-down design is somewhat iterative in nature. After the initial decomposition, each resulting subproblem may require further decomposition until the problem reaches a manageable degree of complexity. Experts proceed with top-down design by selecting appropriate templates for each problem.

In another design technique called stepwise refinement, experts engage in successive restatements of the problem

BEST COPY AVAILABLE

specification with each step closer to machine level notation. The original problem specification describes in natural language a process the computer is to perform. Stepwise refinement means to translate the process description into language the machine understands through incremental stages. Experts can do this well because they are very familiar with the language the machine uses. Experts know the degree of precision and the degree of clarity needed to describe the process for a machine solution. Ultimately, they generate unambiguous statements of their program design.

Characteristics of Students of Programming

Students, those who are just beginning to learn a programming language, usually differ dramatically depending on the sort of instruction they have received (Soloway, Ehrlich, Bonar, & Greenspan, 1982). Members of the ACCCEL project have observed over 25 junior high BASIC programming classes. Most classes offer instruction which emphasizes the language features, and which often fails to provide instruction in how to combine the language features into larger algorithms.

Much of the programming instruction which we observed could be described as drill and practice in learning programming language features. Students are introduced to a language feature such as the PRINT statement. They write programs using that statement. Their understanding of the

BEST COPY AVAILABLE

program is basically at the level of a single line. They type in a line and get feedback about their use of the PRINT statement. Students respond by typing in a different line which hopefully corrects the mistake they have made initially. These students are engaged in drill and practice on a language feature.

Instruction rarely emphasizes the templates which experts use for solving programming problems. Students, therefore, fail to acquire templates to help them decompose problems and plan problem solutions.

Novice programmers are characterized by a "rush to the computer." They frequently attempt to go from a statement of the problem directly to the program code without any consideration of how to design the code. Novices appear to lack the tools necessary for constructing intermediate states between the problem specification and the problem program code. They rarely receive an opportunity to observe their teachers or expert programmers model the use of planning.

The expert process of stepwise refinement is also neither required nor really necessary for most assignments that novices receive in programming courses. It appears that many novices fail to grasp the notion that programs are detailed process descriptions which can be refined out of a natural language description. Instead, novices usually have a tinker toy model of program construction. They presume that programs are assembled by piecing the language features

BEST COPY AVAILABLE

together. They fail to understand that the natural language problem description is less precise and more ambiguous than a problem description in machine terminology. Therefore, they do not engage in the activities required for refining the natural language statement of the problem into a statement which can be decomposed and coded into a problem solution. As a result, when they are asked to solve problems which are more complex than simple translations of known language features, their solutions are often poorly organized, incorrect, or inefficient. Thus, the top down design and stepwise refinement which experts use to write programs is not taught nor required by most introductory programming courses at the junior high level.

ACCCEL Explicitness

The ACCCEL explicitness intervention fosters higher cognitive skill by providing students with some abstract templates which they can use for stepwise refinement of the problem specifications. We set out to provide students with a mechanism for constructing a problem solution that was more detailed than the available problem specification but less detailed than the actual language statements. Thus, we wished to encourage students to consider an intermediate state between the problem specifications and the program code.

As noted above, most students go directly from the

BEST COPY AVAILABLE

problem specification to the terminal. This frequently leads to frustration and inefficient trial and error solutions. However, it should be noted that students are basically very happy when working at the terminal and fail to associate their difficulties in achieving a solution with their lack of planning. Students believe they need more terminal time, not that they need to plan, to solve problems effectively.

Because of the nature of novice instruction in junior high, it was impossible to emphasize planning until students had gained at least a reasonable subset of language features. Thus, we selected an instructional setting where students had already been programming for one semester and had gained enough familiarity with the language that they could be given problems which required planning. Since instruction up to this point had not emphasized planning, our intervention required a considerable departure from previous activities. Our intervention interfered with students' access to the computer terminals.

Thus, our approach was to provide abstract templates which students could use to refine the problem specifications. We also required students to perform numerous exercises where they took problem specifications and translated them into the templates that we provided. We did not require students to code their solutions from the templates, but rather, provided extensive experience in mapping problem

BEST COPY AVAILABLE

specifications onto the templates.

Intervention Procedure

The particular design technique we selected to teach students, structure diagraming, is one of several graphical methods. It is a method for pictorially representing the logical organization of the solution to a problem. The subjects were eighth grade students (age 12 and 13) in an urban school. The students had just begun the second semester of the first course in programming in BASIC. Most of them were familiar with elementary statements in BASIC, but were not yet adept at writing programs.

The instruction consisted of five lessons, each a class period long. The classroom instruction was carried out by two of the authors. Each lesson was comprised of a lecture and student exercises. The lecture was a brief (10-20 min.) explanation about the design technique, focusing on one particular aspect. The exercises were prepared by the project and students worked on them at their desks (not at the computer). The exercises provided examples and problems to be designed.

The first lesson provided a motivating example using a dart game, explained the purpose of the technique, and introduced three basic program structures or blocks: action, loop, and decision (see Figure 1). The assignment involved classifying simple English problem statements according to

BEST COPY AVAILABLE

the three types of blocks.

The second lesson presented a way to draw a diagram which corresponded to each block. The assignment required students to draw diagrams for problems they had classified in the previous exercises.

In the third lesson the diagraming technique was expanded to problems that combined blocks. The assignment involved drawing diagrams with combined block structures from a problem statement.

The fourth lesson proceeded to advanced problems where blocks were nested within one another. Students worked exercises drawing structure diagrams for these complex problem types.

The fifth lesson explained the conventions for translating the structure diagrams into BASIC code. Specific language features were described for coding each block.

Subjects

Participants in this program were 30 eighth graders in a racially and socio-economically mixed classroom. They had just begun the second semester of their first programming course in BASIC. When we instituted the intervention, these students had been introduced to most BASIC language features, including input, output, loops, and decisions.

BEST COPY AVAILABLE

Most were familiar with elementary statements in BASIC but were not yet adept at writing programs. The class was self-paced in the sense that students proceeded through a series of problems at their own speed. The most advanced students were solving the 85th problem which had been offered that year, whereas the least advanced students were still on the 20th problem. These problems were, in general, less complex than those used in our exercises.

Results

As a result of the intervention, all the students in the class were able to use structure diagrams for simple problem specifications. Problems which all students could represent with structured diagrams are given in Table 1.

Students were considerably less successful when they were asked to produce structure diagrams for problem specifications where the superficial grammatical features of the text did not match the block types they had been taught.

We spent considerable effort preparing problems as standardized as possible. They were much less ambiguous than problem specifications in most textbooks. Even with these carefully prepared problems, however, students often failed to identify the salient features that were cues to the selection of the proper block type. As long as the superficial form exactly matched the block types, students were able to proceed. However, if the text of the problem

statement varied from the prototypical form, or required interpretation, most students were at a loss as to how to begin. The problem statements in Table 2 presented difficulties for students. For example, most students did not realize that problem 1 in Table 2 was a looping problem.

Students also had difficulty in determining how much to refine their specifications. Thus, students frequently did not refine the problem specifications to the level of detail of the available block types. For example, problem 3 in Table 2 requires students to write a structure diagram for computing an average. Many students treated the computation of the average as an action. They did not realize that computing an average required a looping process. There was an example problem showing how averages were computed, but apparently the students did not benefit from it. The result was that their solutions were not refined to an appropriate level of detail.

A third kind of student difficulty was a failure to distinguish process from content in the problem statement. For example, in problem 4 in Table 2, students were unable to differentiate between the operations to be performed and the data that the operations were to be performed on. Thus, they did not differentiate between the counting operation required in problem 9 and the data which was essentially the report cards.

Finally, a major difficulty was that students failed to

see the benefits of the structure diagram process for helping them to solve programming problems. Students preferred to be on-line with the computers. A few students employed structure diagrams as an intermediate step between the problem specification and coding of a problem solution. However, these students were in the minority. It seemed that students turned to structure diagrams as a last resort when all other approaches for programming failed. Perhaps, if these students were given challenging assignments for which they could find no other solution, they might attempt to use structure diagrams. This difficulty, as we anticipated, was due to the lack of previous emphasis on planning problem solutions. Our intervention represented a departure from previous practice in this classroom.

After we completed the intervention, but before we administered the postest, serious discipline problems arose in this class. A more stringent class management scheme was adopted after the intervention. The students rebelled against the new state of affairs. Therefore, it was felt that this class was no longer comparable with the other BASIC classes. Further evaluation of the intervention was not pursued.

Discussion

This investigation revealed that students can use structure diagrams for refining problem specifications, although they have difficulty as soon as the features of the

problem specification deviate from those they have already encountered. It seems clear that more varied experience mapping statements onto the structure diagrams is required.

Although students can use structure diagrams, they have difficulty using them when the problem specifications become complex. Similarly, they have difficulty coding solutions for complex problem specifications. Our experience suggests that the difficulties in coding may reflect difficulties in understanding the problem specification. Thus, students need additional instruction both to interpret the problem specifications and to translate these interpretations into program code.

Current modes of instruction fail to communicate the value of planning in programming. This lack of appreciation of planning stems in part from certain characteristics of the instruction. First, students initial programming experiences do not require planning, and therefore, the advantages of planning are not apparent to them. Second, students find the on-line experience very motivating and they prefer to be on-line interacting with the computer, even if they are not making progress in solving problems. It appears that the interactive nature of the computer learning environment has not been well channeled to the higher cognitive skill of planning.

One important reason why students fail to appreciate planning is that many students can solve even the most

difficult problems assigned without planning. The most difficult problem which students were currently attempting was to write a number guesser program. Many students could envision how to solve this problems without spending any time planning.

Our experience in implementing the ACCCEL explicitness treatment suggests some directions that might be considered. First, rather than beginning programming instruction with drill and practice on the language features, it would seem quite appropriate to begin instruction with comprehension of program code. Students could be given reasonable sized programs (10-50 lines of code) and could be encouraged to come to understand those programs. Those programs would demonstrate how planning is used in programming. Students could see how experts use planning to write a big program. Thus, students would have a better understanding of the role of planning in programming. Second, structure diagrams could be used to help students comprehend a large program. A large program could be represented using structure diagrams. Comprehension of the program could be encouraged by using structure diagrams to illustrate the templates used by the programmer to construct the program. Instruction could then proceed by demonstrating the top-down design and the successive refinements used by the expert programmer to construct the program.

Programming instruction has the potential for fostering

the higher cognitive skills called for by the many recent reports on the state of educational practice but, so far, the potential is not being achieved. Instruction which builds a chain of cognitive consequences culminating in the planning skills used by expert programmers requires early and consistent emphasis on these skills. Teachers are needed who can demonstrate planning. Texts are needed which delineate the steps between problem specification and program code. Research is needed to more clearly understand the chain of activities which will facilitate the desired consequences.

TABLE 1

PROBLEMS WHICH MAPPED DIRECTLY
ON TO THE STRUCTURAL DIAGRAMS

1. Print "RAIDERS ARE CHAMPS" 39 times.

2. Compute 7 times 17.

3. If your name is the same as the teacher's, print "SAME", otherwise print "DIFFERENT."

4. For each number from 1 to 50, print the number and its square root.

5. Brian and Jenny need to score a total of 10000 points on Pacman to win the team competition. Input their individual scores, add them together, and print whether they won or not. 6. Ask the user to enter his or her name. For each letter in the name, print a star.

TABLE 2

PROBLEMS WHICH WERE DIFFICULT
TO MAP ON TO THE STRUCTURAL DIAGRAMS

1. Sally has 50 dollars in the bank. She plans to deposit 5 dollars each month. For each month of the year, calculate her new account balance.

2. Joe can read 65 pages a day. Compute how long it would take him to finish a 350 page book.

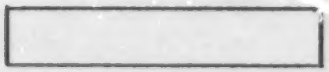
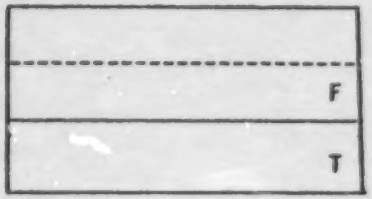
3. Find the average height of students in your class. If the average is grater than 60 inches print "Giants", otherwise print "Shrimps."

4. Given a list of classes and grades from your report card; reach each grade and if the grade is an A, add one to a total. (Count the number of A's on your report card.)

BEST COPY AVAILABLE

Figure 1

PROBLEM ANALYSIS

	ACTION	LOOP	DECISION
DEFINITION:	An Action is one or more instructions that the computer performs in sequential order.	A Loop is one or more instructions that the computer performs repeatedly.	A Decision is making a choice among several actions.
DIAGRAM:			
EXAMPLE:	Print "Jack" or Add 31 plus 17	For each Row on the screen Print your name	If temperature > 75 Print "warm" F Print "hot" T
	An Action block is a rectangle drawn around the action to be performed.	A Loop block shows the loop "control" in the top part, and the repeated part or "body" in the enclosed box.	A Decision block shows the condition being tested in the top part (above the dotted line). The false (F) alternative is shown next, and the true (T) alternative is shown at the bottom.

BRAINARD COBOL MANUAL

BIBLIOGRAPHY

Atwood, M. E. and Ramsey, H. R. Cognitive structures in the comprehension and memory of computer programs: An investigation of computer debugging. Tech Report TR-78-A210, U. S. Army Research Institute for the Behavioral and Social Sciences, Alexandra, VA, 1978

Becker, H. J. School uses of microcomputers. 1-5, Baltimore, MD.: Center for Social Organization of Schools, The Johns Hopkins University, 1984.

Brooks, R. Studying programmer behavior experimentally: The problems of proper methodology. Communications of the ACM, 1980, 23(4), 207-213.

Heller, J. I. and Reif, F. Prescribing effective human problem-solving processes: Problem description in physics. Berkeley, CA: Lawrence Hall of Science, 1980. Paper presented at the annual meeting of the American Educational Research Association in New York, March 1982

Jeffries, R., Turner, A. T., Polson, P. G., and Atwood, M. E. Processes involved in designing software. In J. R. Anderson (Ed.), Cognitive skills and their acquisition. Hillsdale, NJ: Lawrence Erlbaum Associates, 1981.

Kurland, D. M. and Pea, R. D. Children's mental models of recursive Logo programs . (Tech. Rep. No. 10), New York: Bank Street College of Education, Center for Children and Technology, 1983.

Larkin, J., McDermott, J., Simon, D. P., and Simon, H. A. Expert and novice performance in solving physics problems. Science, June, 1980, 208, 1335-1342.

Linn, M. C. Fostering equitable consequences from computer learning environments. Lawrence Hall of Science, University of California, Berkeley, CA: Project ACCCEL, 1984.

Linn, M. C. and Fisher, C. W. The gap between promise and reality in computer education: planning a response. Proceedings of Making Our Schools More Effective: A Conference for Educators, 1983, San Francisco, CA: Far West Laboratory.

Norman, D., Gentner, D., and Stevens, A. Comments on learning schemata and memory representation. Cognition and instruction, 1976, Hillsdale, NJ: Erlbaum.

Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J. What do novices know about programming?. Directions in human-computer interactions, 1982, Norwood, NJ: Ablex.

BEST COPY AVAILABLE

DISCUSSION

Jan Hawkins

Center for Children and Technology
Bank Street College of Education

The teaching and learning of programming has quite rapidly become an important topic among educators. For years, programming was taught in a small number of schools as a specialized skill for vocation. It has recently assumed importance as a central curriculum topic for all students in many schools, beginning at the elementary level. This is due in part to the flood of microcomputer technology into schools, but also to the notion that programming is a learning environment where students will acquire general problem solving skills. This is a powerful reason for the rapid assimilation of programming into the curriculum. Without full commitment to the importance of programming as a skill for all students, even the possibility that its practice allows students to acquire broadly applicable skills for solving problems is a reasonable rationale for its adoption. This rationale is particularly timely today since schools are widely concerned with evidence that many students fail to learn problem-solving skills.

Two major questions are raised in this symposium, and interleaved throughout the papers. First, how can programming be most effectively taught? Second, what is learned when students engage in programming practice--both with respect to programming commands and concepts, and to broader problem-solving skills? Not surprisingly, as these authors look more closely at programming practices and possible circumstances of transfer, the situation quickly becomes more complex than at initial probing.

Specifically, various sources of complexity are identified in the papers. As the first two papers point out (Kurland, Clement, Mawby & Pea; Kurland, Mawby & Cahir) in order to determine the effects of programming and decide about the best conditions for learning, a better understanding of just what activities programming entails is needed. Kurland et al note that programming is a multifaceted activity, not a unitary topic. The task varies according to language, programming environment, and social surround. Under the description "programming" two students may be doing radically different tasks, and learning quite different skills. In addition, as programming arrives in the classroom, the goals of the activity are not clear. What are the conditions of expertise? What should we expect proficient students to be able to do or know? Neither of these issues has an easy solution. Kurland et al emphasize the importance of planning a program, of understanding the structural aspects of the activity, as opposed to the mere production of code. This preparatory work and understanding of the broad scheme seems to characterize the work of experts, and contrasts sharply with the work of students who participated in the research.

Dalbey, Tourniaire, and Linn also identify some of the characteristics and techniques of expert programmers (e.g. programming templates, procedural skills), and attempt to embody one of these planning techniques in the programming curriculum of eighth grade novice programmers. In contrast to an instructional approach which emphasizes language features, they provide instruction in structured diagramming--a representational format for planning a program prior to beginning coding. While all students were able to learn this technique, many had difficulty transferring the technique to novel problem types, and many failed to see

the value of this activity. The issue of transfer of skill, then, is complex even within the domain of programming problems. In addition, the techniques valued by experts may have a complex relationship to the types of programming supports that are effective with novices. The identification and adoption of expert techniques may not be the critical factor for novices. Understanding the conditions in which students perceive a programming tool as useful for problem solution would seem essential in the building of an effective instructional sequence.

While Erlich, Abbott, Salter and Soloway find some evidence of transfer of procedural programming skills to non-programming algebra problems, they also note the intricacies of data interpretation. The influence of gender relative to programming experience they document is not readily understandable in the improvements noted among male programming students. These sorts of gender differences have been widely noted in mathematics achievements (cf. Brush, 1980), and with computers (cf. Hawkins, 1984). The evidence of gender differences is not surprising, as performances in these domains seems complexly related to differential expectations and performance conditions. These results add an interesting note of complexity to the understanding of conditions of transfer.

Pea suggests that research concerning the transfer of skills must be based on a thorough understanding of the functional activities of a "culture". How does the group enlist particular skills in the carrying out of internally meaningfully activities? Noting the findings of a research program that failed to find evidence of improvement in planning skills among elementary students after a year's programming experience, Pea recommends both changes in the instructional environment and in the

places one looks to find generalization of ability. Programming per se is not a privileged environment in which general skills are "naturally" acquired. Rather, bridges must be built between the programming work and other problems in a learning setting mutually constructed by teachers and students.

These studies take on the difficult task of examining the conceptual and policy questions raised by the enthusiasm in the educational community for programming. In addition to the implications of their findings, the papers articulate paths for yet deeper probing. Among the most salient:

What are the goals of programming work for students at different levels? One angle of approach was adopted by Kurland et al: what are experts able to do? This information about end-state, or expert practice, should help to shape the programming environment that students encounter. Additional analyses of the characteristics of expertise are essential. Complementary approaches are also necessary. The development of goals is an issue of values and priorities in a larger cultural surround. As teachers rightly note, when programming comes into the curriculum for all students, something else goes out. In light of the time commitment and evidence of possible prerequisite abilities, is the pursuit of programming expertise a reasonable goal appropriate for all students? If it is a privileged environment for learning problem-solving skills, its adoption can be well justified. The emerging evidence, however, is equivocal and the conditions of transfer are certainly more complex than initial claims imply.

An analogy to the historical conditions of written literacy is useful in thinking about the goals for programming. Resnick and Resnick (1977) trace the historical development of definitions of literacy in terms of

reading comprehension. An examination of changing criteria for literacy in Europe and American reveals that sharp shifts occurred, relative to social conditions and valued skills. Our current definition of literacy is a recent one (at most three generations old)--from memorization and reading in order to decipher, to reading in order to comprehend and develop problem-solving capacities; from education of an elite, to mass literacy. The current criterion defines reading for comprehension as a standard for instruction, and adopts notions of functional literacy for participation in today's society as goals (e.g. ability to read a newspaper, fill out a form, read an instructional manual).

Although programming is a much more recent cognitive technology, underlying some of the curriculum approaches is a notion of general, mass "literacy" in these skills. The development of goals for this area, I believe, rests not simply on an analysis of the abilities of experts, but also on a analysis of necessity and meaning of functional programming literacy in the broader society. In the rush to develop a programming curriculum, this analysis for education has yet to be done.

Where does one look for effective instructional variables? There is a nice correspondence in these papers between promising conditions of instruction for programming, and possible conditions of generalization. There is a focus on the need to explore not simply the means for instructing primitive commands of a programming language and the practice of coding, but on the need to teach planning and structure in programming concepts. This then points to a need to explore programming environments (e.g., editors, planning aids, pseudo code, debugging assistants, trace facilities), and means for encouraging students to think carefully about the problem and the overall structure of

the program prior to entering code. Kurland et al, and Dalbey et al have already begun work in this direction. Their results indicate the need for more. Rather than simply comparing the relative advantages of languages, additional research concerning the effects of various representations and programming supports for novices is required. Perhaps it is through structured supports in programming problems that generalization can be seen by students, and therefore by researchers.

How else can transfer be examined? Thus far, examinations of transfer of programming skill have largely been based on an abstract analysis of the skills entailed in accomplishing a working program (cf. Pea & Kurland, 1984). Thus, one looks for transfer of procedural or planning abilities that are assumed to be evoked in programming practice. One develops tasks that are intended to evoke these skills--the major requirement of such tasks is that component skills are invited in performance. The quality of such skills in students' performances is then assessed. However, as Pea points out, programming is a functional activity like other skilled performances of a culture. It takes place in particular circumstances for particular reasons. Relatively little attention has been paid thus far to the types of problems or programming goals that students are asked to engage in. Programming practices were developed in order to do particular classes of tasks better or differently. Programming is a topic, but it is also a particular medium or means to solve problems that have been identified as appropriate.

It might be useful, therefore, to pay careful attention to the types of problems students are asked to find solutions to through the medium of programming. Many of the these tasks would have previously been done through different representations and procedures with different media.

(e.g. math problems done with paper, pencil and a calculator). Rather than designing tasks with the focus on structural similarities to programming, tasks that are functionally similar to students' programming work might evoke a different approach that has been learned through programming. The representations and skills practiced through use of the technology to solve particular problems might result in new ways of seeing those classes of problems, independent of the technology.

What are effective social conditions for learning to program? There is evidence in these studies that social circumstances play an important role in the programming performances we analyze. This is noted at two levels in these papers. In terms of the larger culture, functional activities are important in defining how skills are acquired and applied. In addition, gender--the expectations and orientations to performance in particular circumstances--may be an important variable in learning and for assessing skill. Cognitive questions about development, and educational questions about implementation must take account of these larger social circumstances. More needs to be known.

At a more immediate level, the locus of programming instruction is the classroom. Classrooms are complex social settings in all grades. A programming curriculum exists within a well-established set of expectations--instruction and interaction among classroom members. In establishing an effective programming agenda for schools, more needs to be known about the social structure and resources that embody the ideas. For example, Kurland mentions the example of "code stealing", where one student borrows sections of code to construct a program--code that he does not fully understand. If this reliance on others' work is used appropriately, it is possible to incorporate the exchange of work as an

effective instructional practice. The student, for example, could be encouraged to construct variations on the program which would require changes in the code. Whether he knew how to vary it himself, or knew where to get help, providing situations which focus on variations in construction and use of a particular section of code could lead to improved understanding.

These papers are important in further defining the nature of problems in developing effective instructional programs for programming skills, with an emphasis on using this environment to teach problem solving. They thus point to important paths for future research.

REFERENCES

Brush, L. E. (1980). Encouraging girls in mathematics. Cambridge, MA: Abt Books.

Hawkins, J. (1984). Computers and girls: Rethinking the issues. New York: Bank Street College of Education, Center for Children and Technology Technical Report No. 24.

Pea, R. D. & Kurland, D. M. (1984). Towards cognitive technologies for writing. Unpublished manuscript, New York: Bank Street College of Education, Center for Children and Technology.

Resnick, D. P. & Resnick, L. B. (1977). The nature of literacy: An historical exploration. Harvard Educational Review, 47(3), 370-385.